UNIVERSIDADE FEDERAL DE SERGIPE

CENTRO DE CIÊNCIAS EXATAS E TECNOLÓGICAS

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

SDNMonitor: Um Serviço de Monitoramento de Tráfego em Redes Definidas por Software

Emanuel Ferreira da Silva

São Cristóvão

2016

UNIVERSIDADE FEDERAL DE SERGIPE

CENTRO DE CIÊNCIAS EXATAS E TECNOLÓGICAS

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Emanuel Ferreira da Silva

SDNMonitor: Um Serviço de Monitoramento de Tráfego em Redes Definidas por Software

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação (PROCC) da Universidade Federal de Sergipe (UFS) como parte de requisito para obtenção do título de Mestre em Ciência da Computação.

Orientador: Prof.ª Dr.ª Edilayne Meneses Salgueiro

Coorientador: Prof. Dr. Ricardo José Paiva de Britto Salgueiro

São Cristóvão
2016

Emanuel Ferreira da Silva
SDNMonitor: Um Serviço de Monitoramento de Tráfego em Redes Definidas por Software/ Emanuel Ferreira da Silva. – São Cristóvão, 2016-
79 p. : il. (algumas color.) ; 30 cm.

Orientador: Prof.ª Dr.ª Edilayne Meneses Salgueiro

Dissertação (Mestrado) – UNIVERSIDADE FEDERAL DE SERGIPE
CENTRO DE CIÊNCIAS EXATAS E TECNOLÓGICAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO, 2016.

1. Redes Definidas por Software. 2. Monitoramento de Tráfego. 3. Balanceamento de Carga. 4. OpenFlow. I. Prof.ª Dr.ª Edilayne Meneses Salgueiro. II. Universidade Federal de Sergipe. III. Programa de Pós-graduação em Ciência da Computação. IV. SDNMonitor: Um Serviço de Monitoramento de Tráfego em Redes Definidas por Software

CDU 02:141:005.7

Emanuel Ferreira da Silva

SDNMonitor: Um Serviço de Monitoramento de Tráfego em Redes Definidas por Software

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação (PROCC) da Universidade Federal de Sergipe (UFS) como parte de requisito para obtenção do título de Mestre em Ciência da Computação.

BANCA EXAMINADORA

Prof.ª Dr.ª Edilayne Meneses Salgueiro, Presidente
Universidade Federal de Sergipe (UFS)

Prof. Dr. Ricardo José Paiva de Britto Salgueiro, Membro
Universidade Federal de Sergipe (UFS)

Prof. Dr. Edward David Moreno Ordonez, Membro
Universidade Federal de Sergipe (UFS)

Prof. Dr. Kelvin Lopes Dias, Membro
Universidade Federal de Pernambuco (UFPE)

SDNMonitor: Um Serviço de Monitoramento de Tráfego em Redes Definidas por Software

Este exemplar corresponde à redação da Dissertação de Mestrado, sendo a defesa do mestrando **Emanuel Ferreira da Silva** para ser aprovada pela banca examinadora.

Trabalho aprovado. São Cristóvão, 30 de agosto de 2016:

Prof.ª Dr.ª Edilayne Meneses Salgueiro
Orientador

Prof. Dr. Ricardo José Paiva de Britto Salgueiro
Membro

Prof. Dr. Edward David Moreno Ordonez
Membro

Prof. Dr. Kelvin Lopes Dias
Membro

Resumo

Com a popularidade da Internet e a emergência de novos serviços, tornou-se cada vez mais necessário fazer um planejamento da rede, com o objetivo de assegurar que cada um dos elementos que a compõem sejam utilizados de forma eficiente. Além disso, é necessário controlar e monitorar a rede, verificando se tudo está sendo executando da maneira que foi planejada. Em redes que utilizam o paradigma SDN, através da introdução de um controlador de rede, é possível a separação entre o plano de dados (hardware) e o plano de controle (software) existentes nos dispositivos de rede, permitindo que novos protocolos e novas tecnologias sejam implementadas e testadas em qualquer dispositivo de rede, independente do seu fabricante. Em contrapartida, surge o seguinte questionamento: como aplicar monitoramento de tráfego em uma rede SDN diante da sua arquitetura de controle centralizada sem causar atrasos ou inconsistências? Este trabalho propôs um serviço de monitoramento de tráfego em redes SDN baseado no protocolo OpenFlow, chamado SDNMonitor, que tem como principal objetivo prover uma visão do tráfego de dados da rede em três níveis de granularidade, por cada porta de cada *switch*, por cada fluxo e por cada serviço de rede. Adicionalmente, também foi proposto um serviço de balanceamento de carga, baseado na utilização dos algoritmos Round-Robin e *Bandwidth-Based*. A avaliação experimental destes serviços foi realizada através de experimentos controlados, onde foram gerados e monitorados alguns tráfegos. Os resultados demonstraram que o serviço SDNMonitor conseguiu monitorar o tráfego da rede nos três níveis de granularidade sem impactar de forma negativa no seu funcionamento, e que o serviço de balanceamento de carga foi capaz de melhorar o tráfego da rede.

Palavras-chaves: Redes Definidas por Software. Monitoramento de Tráfego. Balanceamento de Carga. OpenFlow.

Abstract

With the popularity of the Internet and the emergence of new services, it has become increasingly necessary to make a network planning, in order to ensure that each of the elements that compose it are used efficiently. Moreover, it is necessary to control and monitor the network, making sure that everything is running the way it was planned. In networks using the SDN paradigm, by introducing a network controller, is possible the separation between the data plane (hardware) and control plane (software) existing on the network devices, allowing that new protocols and technologies are implemented and tested on any network device, regardless of its manufacturer. In contrast with it, the following question arises: how to apply traffic monitoring in an SDN network ahead of his centralized control architecture without causing delays or inconsistencies? This work proposed a traffic monitoring service for SDN networks based on the OpenFlow protocol, called SDNMonitor, which aims to provide a view of network data traffic at three levels of granularity, for each port of each switch, for each flow and for each network service. Additionally, it was also proposed a load balancing service based on the use of Round-Robin and Bandwidth-Based algorithms. The experimental evaluation of these services was conducted through controlled experiments, which were generated and monitored some traffic. The results showed that the SDNMonitor service could monitor the network traffic in the three levels of granularity without impacting negatively on its operation, and that the load balancing service has been able to improve network traffic.

Key-words: Software-Defined Networking. Traffic Monitoring. Load Balancing. OpenFlow.

Lista de figuras

Lista de algoritmos

Lista de tabelas

Lista de abreviaturas e siglas

API	Application Programming Interface (Interface de Programação de Aplicativo)
CLI	Command Line Instructions (Instruções de Linha de Comando)
CORBA	Common Object Repository Broker Architecture (Arquitetura de Barramento de Repositório de Objeto Comum)
CPU	Central Processing Unit (Unidade Central de Processamento)
CSS	Cascading Style Sheets (Folhas de Estilo em Cascata)
DER	Diagrama Entidade Relacionamento
EUA	Estados Unidos da América
HTML	HyperText Markup Language (Linguagem de Marcação de Hipertexto)
IP	Internet Protocol (Protocolo de Internet)
JSON	JavaScript Object Notation (Notação de Objetos JavaScript)
MAC	Media Access Control (Controle de Acesso ao Meio)
NBAPI	Northbound Application Programming Interface (Interface de Programação de Aplicativo Northbound)
NFV	Network Function Virtualization (Virtualização da Função de Rede)
ONF	Open Networking Foundation
OSGI	Open Services Gateway Initiative (Iniciativa Gateway de Serviços Abertos)
PHP	HyperText Preprocessor (Pré-processador de Hipertexto)
REST	Representational State Transfer (Transferência de Estado Representacional)
SBAPI	Southbound Application Programming Interface (Interface de Programação de Aplicativo Southbound)
SDN	Software-Defined Networking (Rede Definida por Software)
SGBD	Sistema de Gerenciamento de Banco de Dados

SNMP	Simple Network Management Protocol (Protocolo Simples de Gerenciamento de Rede)
SQL	Structured Query Language (Linguagem de Consulta Estruturada)
TCP	Transmission Control Protocol (Protocolo de Controle de Transmissão)
UDP	User Datagram Protocol (Protocolo de Datagrama do Usuário)
VLAN	Virtual Local Area Network (Rede Local Virtual)
WBEM	Web-Based Enterprise Management (Iniciativa de Gerenciamento Baseado em Web)
XML	eXtensible Markup Language (Linguagem de Marcação Extensível)

Sumário

1 Introdução

O controle distribuído e a adoção de uma pilha de protocolos padrão permitiu que a Internet se tornasse um meio de comunicação essencial. Esse controle distribuído é implementado nos elementos de rede, tais como *switches* e roteadores. Esses elementos de rede têm a tarefa de inspecionar cada novo pacote de dados que chega em uma interface e repassar para a interface mais adequada, atendendo os requisitos de um determinado serviço, usuário ou do provedor. Essa tarefa de inspeção e repasse pelos elementos da rede é um problema computacional complexo (DENAZIS et al., 2015) que envolve o gerenciamento de um grande número de elementos.

Novas tecnologias devem acompanhar o aumento da necessidade de gerenciamento dos recursos da rede. A diversidade de serviços e o aumento do acesso à Internet requer um planejamento cuidadoso da rede. Cada elemento da rede é um recurso e para que esse recurso seja utilizado eficientemente, informações sobre o comportamento do tráfego são necessárias para a tomada de decisão. A tomada de decisão, visando melhorar o desempenho enquanto se adapta às condições da rede é um cenário desafiador para as arquiteturas atuais.

Alguns autores como Papadimitriou et al. (2009) afirmam que nas redes tradicionais é possível perceber que existe um engessamento, conhecido como "ossificação" da rede, visto que os elementos de rede atuam como "caixas-pretas" devido ao fato de possuírem hardware e software proprietários. Esse engessamento acarreta falta de flexibilidade para que novas tecnologias sejam criadas e testadas por pesquisadores que não fazem parte da indústria, pois sua implementação fica sob a responsabilidade dos fabricantes dos elementos de rede.

De acordo com (DIEKMANN, 2013), cenário semelhante era visto nas arquiteturas de computadores, como pode ser visto na Figura 1.1(a), onde as aplicações, os sistemas operacionais e o hardware eram proprietários, o que dificultava a criação de novas aplicações, visto que elas deveriam ser feitas para um sistema operacional específico em um hardware específico. Ou seja, para que elas funcionassem em outra máquina, teriam que ser modificadas ou mesmo reescritas. Para mudar essa concepção, foi necessário realizar uma separação entre hardware, software e aplicação, criando interfaces de comunicação entre eles, dessa forma, sistemas operacionais distintos passaram a executar em componentes de hardware distintos, assim como as aplicações com relação aos sistemas operacionais, como mostra a Figura 1.1(b).

O paradigma das Redes Definidas por Software (do inglês *Software-Defined Networking*, ou simplesmente SDN) propõe algo semelhante ao que ocorreu nas arquiteturas computacionais, ao cenário das redes de computadores (ver Figura 1.2), trazendo novas possibilidades e permitindo resolver problemas das redes atuais (SHALIMOV et al., 2013).

A principal proposta do paradigma SDN consiste em retirar toda a lógica de controle dos elementos de rede e colocar em um software de forma centralizada. Com isso, as decisões sobre

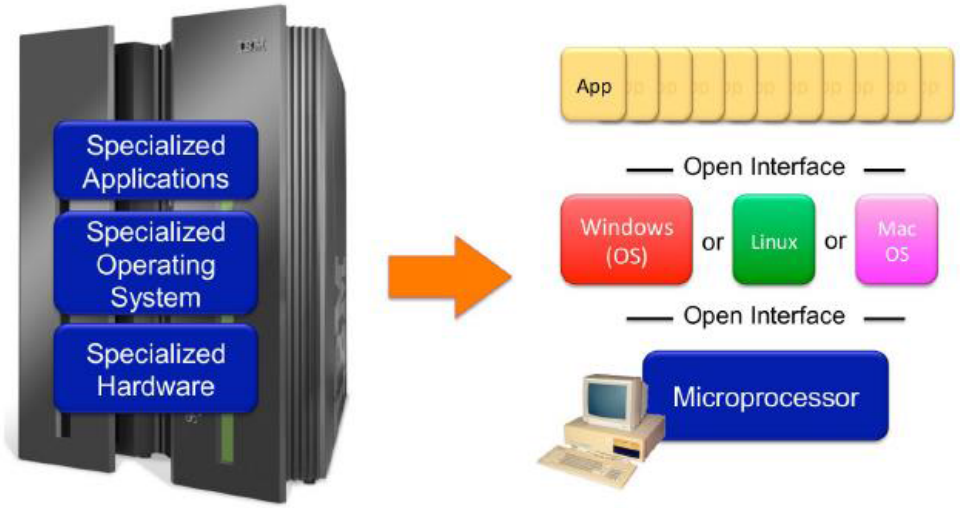


Figura 1.1 – Arquitetura computacional proprietária (a) e arquitetura computacional aberta (b). Fonte: (ART, 2014)

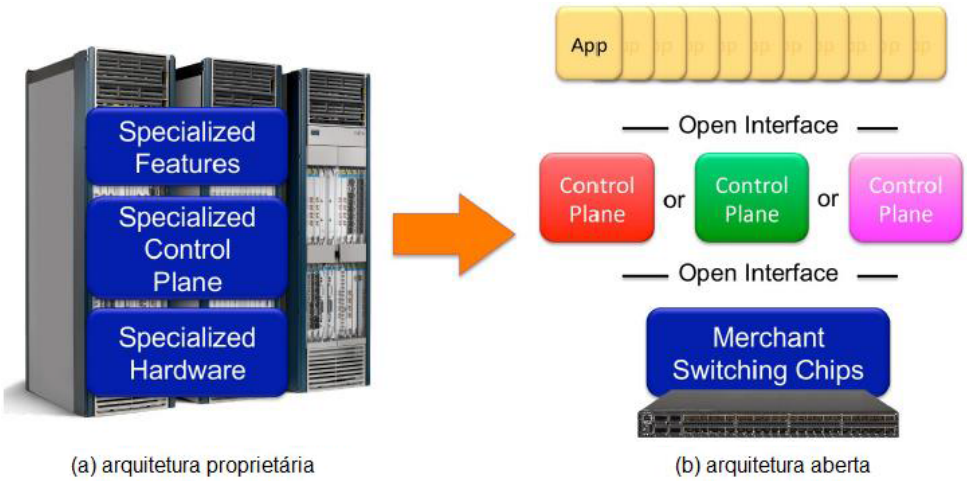


Figura 1.2 – Arquitetura de rede proprietária (a) e arquitetura de rede aberta (b). Fonte: (ART, 2014)

como os pacotes devem ser encaminhados ou descartados são tomadas pelo controlador, sendo repassadas para os elementos de rede que simplesmente as executam. Para que isso seja feito, é feita a separação do plano de dados (hardware) do plano de controle (software) existentes nos elementos de rede (JAIN; PAUL, 2013). Essa separação possibilita a automação da configuração da rede.

Os principais componentes da arquitetura SDN são o controlador, os elementos de rede e um protocolo de comunicação (JAIN; PAUL, 2013). Esses componentes são apresentados na Figura 1.3. O controlador é responsável por centralizar toda a informação da rede, além de determinar os fluxos de todos os pacotes. O controlador também provê uma API (*northbound* API, ou NBAPI) para que aplicações possam especificar requisitos de rede. Os elementos (ou

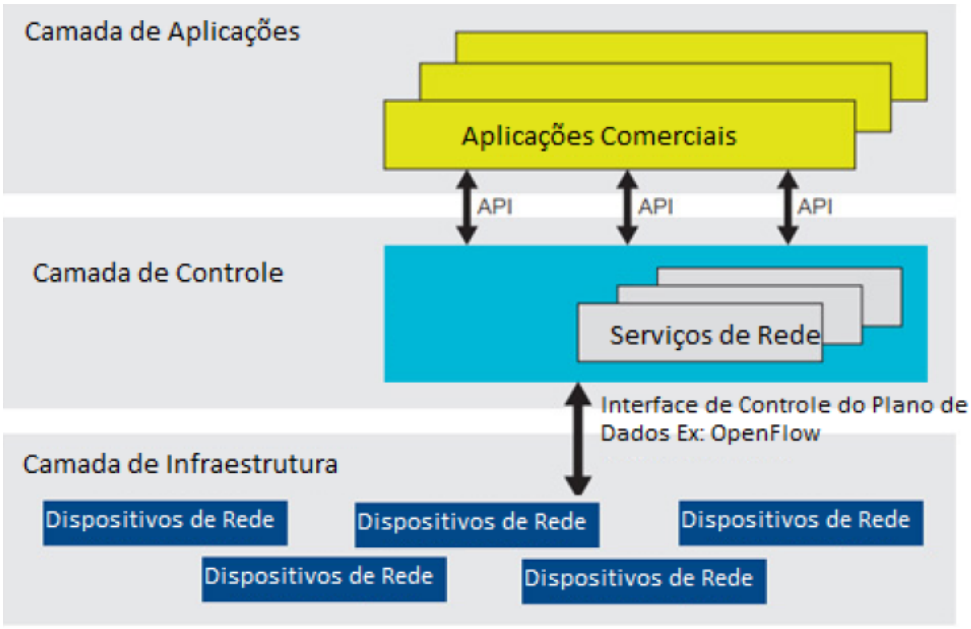


Figura 1.3 – Arquitetura de uma rede SDN. Fonte: (ONF, 2012)

dispositivos) de rede são responsáveis por realizar o encaminhamento dos pacotes na rede, além de proverem uma API (*southbound* API, ou SBAPI) para que o controlador possa enviar as instruções que alimentam suas tabelas de fluxos. O protocolo é responsável por fazer a comunicação entre controlador e elementos de rede.

A *Open Networking Foundation* (ONF) é um consórcio de empresas criado em 2011, nos EUA, para promover a adoção da arquitetura SDN. A ONF promove a adoção do OpenFlow como protocolo para redes SDN (ONF, 2015). O OpenFlow foi inicialmente proposto por (MCKEOWN et al., 2008) para a introdução de SDN de caráter experimental em redes operacionais. Desde então, devido a ampla divulgação e adoção desse protocolo nos elementos de rede, Phemius e Bouet (2013) afirmaram que corre-se o risco de que Openflow e SDN sejam tomados como termos sinônimos. No entanto, o Openflow constitui o protocolo implementado em *switches* e roteadores que possibilita que estes sejam configurados e monitorados pelos controladores.

Segundo ONF (2014), pode ser considerado como um princípio chave da arquitetura SDN, o fato de que aplicações possam monitorar o estado da rede, ou seja, obter informações sobre o tráfego de dados para que possam se adaptar a rede. Além disso, as aplicações se tornam capazes de especificarem requisitos de rede, como *throughput*, atraso, variação do atraso ou disponibilidade.

1.1 Problema

O monitoramento do tráfego de uma rede se mostra como algo importante, visto que permite as aplicações se adaptarem às condições da rede, fazendo um monitoramento de tráfego adaptativo (CASADO; FOSTER; GUHA, 2014). No caso das redes SDN, o monitoramento

de tráfego é possível devido ao fato de que cada regra da tabela de fluxos possui contadores associados, que mantêm o controle de dados estatísticos básicos, como o número e o tamanho total de todos os pacotes processados através de cada uma (CASADO; FOSTER; GUHA, 2014). Os controladores podem fazer a leitura destes dados estatísticos através de mensagens do protocolo OpenFlow.

Em (SUH et al., 2014) também foi abordada a forma como é feita a coleta de dados estatísticos com protocolo OpenFlow para o monitoramento de tráfego. É dito que o protocolo OpenFlow permite ao controlador que sejam coletados dados estatísticos sobre o tráfego da rede, através de contadores locais (ou seja, armazenados no próprio elemento de rede) de pacotes para cada regra da tabela de fluxos, e para cada porta do elemento de rede. Nesse caso, sempre que um pacote se encaixar em uma determinada regra, o contador desta regra é incrementado, e sempre que um pacote passa por uma porta do *switch*, o contador dessa porta também é incrementado.

Segundo Giotis et al. (2014) esta abordagem é muito dispendiosa, pelo fato de que para fazer uma análise estatística sobre o tráfego da rede, o controlador precisa ter uma visão global de toda a rede, e para isso teria que solicitar a todos os elementos as suas informações estatísticas, o que pode causar uma sobrecarga na rede.

Tendo em vista a importância de monitorar o estado da rede através da coleta e análise de dados relacionados ao tráfego, fazer isso em uma rede OpeFlow se mostra um grande desafio devido à forma como são coletados estes dados. Diante disto e da arquitetura centralizada no controlador, atualizar as informações sobre o estado da rede pode gerar atrasos ou inconsistências.

1.2 Hipótese

Diante desta problemática, surge a hipótese de que um serviço de monitoramento de tráfego em redes SDN, seja capaz de monitorar e inclusive alterar o estado da rede, sem impactar de forma negativa no seu desempenho.

1.3 Objetivo

À luz desta hipótese, este trabalho apresenta como objetivo geral desenvolver um serviço de monitoramento de tráfego para redes SDN, sendo necessário atingir os seguintes objetivos específicos:

- Desenvolvimento de um mecanismo de coleta de dados de tráfego;
- Desenvolvimento de uma interface WEB amigável;
- Desenvolvimento de um mecanismo de balanceamento de carga.

1.4 Organização do Trabalho

Este trabalho se encontra dividido da seguinte forma: No Capítulo 2 são apresentados os principais conceitos do paradigma SDN, assim como os principais elementos que compõem a sua arquitetura. O Capítulo 3 apresenta alguns conceitos do monitoramento de tráfego no cenário que precede o paradigma SDN, e introduz o monitoramento de tráfego no cenário do paradigma SDN. Uma revisão bibliográfica é apresentada no Capítulo 4. No Capítulo 5 é descrita a solução proposta para o problema do monitoramento de tráfego em Redes Definidas por Software. Um Estudo de Caso é apresentado no Capítulo 6 com o objetivo de validar a solução proposta. Por último, o trabalho é concluído no Capítulo 7, com uma reflexão do que foi apresentado.

2 Redes Definidas por Software

O principal objetivo do paradigma das Redes Definidas por Software consiste em criar uma arquitetura de rede flexível através da separação entre o (i) plano de dados (ou plano de encaminhamento) – responsável por realizar a comutação e repasse dos pacotes na rede de acordo com determinados fluxos e o (ii) plano de controle – faz uso de uma série de protocolos para criar os fluxos que são encaminhados para os componentes que armazenam o plano de dados.

2.1 Controlador

O controlador integra o plano de controle e é peça fundamental na arquitetura SDN, visto que ele é o centralizador de toda a informação da rede, sendo o responsável por determinar os fluxos de todos os pacotes. Segundo Jain e Paul (2013), ele é responsável por prover uma API (*northbound* API, ou NBAPI) para que aplicações possam especificar requisitos de rede. Fazendo uma analogia com a arquitetura computacional, essa API é semelhante às chamadas de sistema que os sistemas operacionais fornecem para que aplicações possam ser desenvolvidas. Para se comunicar com os elementos de rede, o controlador utiliza uma API chamada de *southbound* API, ou simplesmente SBAPI, provida pelos próprios elementos de rede (JAIN; PAUL, 2013). Novamente fazendo uma analogia com a arquitetura computacional, essa API assemelha-se à que é fornecida pelo hardware para que o sistema operacional possa se comunicar com ele. Devido a essas características, o controlador é visto como um sistema operacional de rede.

Conforme (HU et al., 2014), o plano de controle de uma SDN pode apresentar duas configurações:

1. Centralizado - onde existe somente um controlador responsável por todos os elementos de rede;
2. Distribuído - onde existem vários controladores, e cada um fica responsável por parte da rede, ou seja, por alguns elementos de rede específicos.

Na abordagem centralizada existe o problema de que, se o controlador parar de funcionar, toda a rede será comprometida, principalmente se estiver operando no modo reativo, onde os elementos de rede não possuem uma pré-configuração. A Figura 2.1(a) dá um exemplo de como seria o controle centralizado, onde todos os *switches* dependem do mesmo controlador.

Já na abordagem distribuída, se um controlador parar de funcionar, somente parte da rede será comprometida. A Figura 2.1(b) dá um exemplo de como seria o controle distribuído.

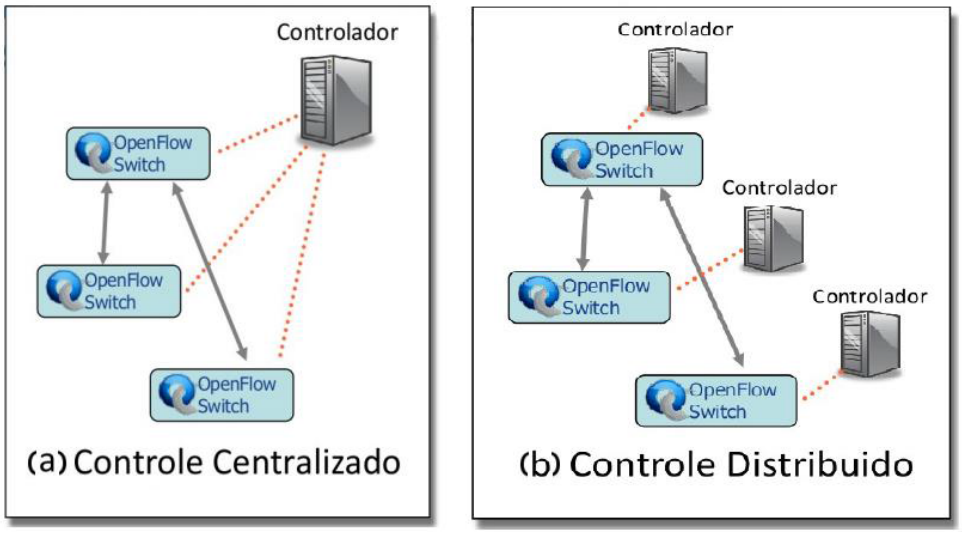


Figura 2.1 – Plano de controle centralizado (a) plano de controle e distribuído (b). Fonte: (RAJASRI, 2011)

Os controladores também podem ser organizados de forma hierárquica, onde um controlador é responsável por um determinado grupo de controladores, um exemplo de controlador hierárquico é o Flowvisor (SHERWOOD et al., 2009). Desenvolvido na linguagem de programação Java, o Flowvisor foi proposto como um controlador capaz de gerenciar redes virtuais, fazendo a ponte entre elas e a rede física.

Uma questão interessante sobre os controladores é que eles podem fazer uso de recursos de alto nível, como LDAP e bancos de dados, para auxiliá-los em sua função, como por exemplo, montagem da topologia da rede ou determinação de qual fluxo um pacote deverá seguir. Atualmente, a maioria dos controladores é desenvolvida em linguagens de programação de propósito geral (ou seja, linguagens que não são específicas para SDN), como Java, Phyton e C, mas também existem pesquisas voltadas para a criação de linguagens de propósito específico (ou seja, linguagens com o propósito de serem utilizadas somente na criação de controladores SDN), como Frenetic e Flowlog (NELSON et al., 2014).

Para se construir um controlador, é necessário determinar qual será o seu modo de operação. Até o momento, existem dois modos de operação, são eles: pró-ativo e reativo (EDWARDS et al., 2014) (ver Figura 2.2).

No modo de operação pró-ativo, as regras de fluxo são pré-configuradas pelo administrador da rede no controlador, sendo repassadas para os elementos de rede. As etapas desse modo de configuração podem ser vistas na Figura 2.2(a). São elas: (0) o controlador envia as regras de fluxo para o elemento de rede (nesse caso, um *switch* OpenFlow), e quando (1) chega um novo pacote no elemento de rede, ele (2) analisa as regras para esse pacote a fim de verificar se ele se enquadra em alguma delas. Em caso afirmativo, é (3) feito o repasse do pacote; caso contrário, (4) o pacote é descartado.

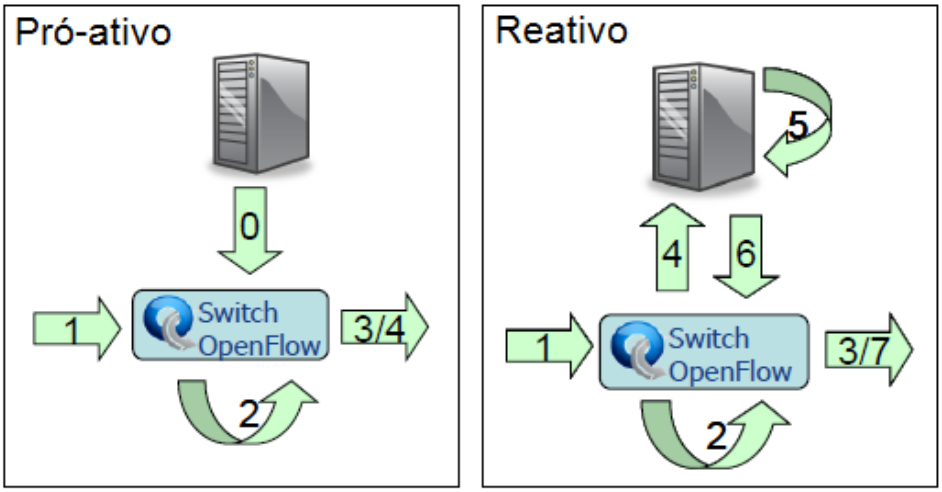


Figura 2.2 – Modo de operação pró-ativo (a) e modo de operação reativo (b). Fonte: (ROJAS, 2013)

Diferentemente do modo de operação Pró-ativo, não existe uma pré-configuração das regras. No modelo de operação Reativo, as regras são criadas de forma dinâmica pelo controlador, de acordo com os pacotes que chegam nos *switches*. As etapas do modo de operação Reativo podem ser vistas na Figura 2.2(b). São elas: (1) chega um novo pacote no elemento de rede, então ele (2) verifica se o pacote se enquadra em alguma das regras existentes; em caso afirmativo, (3) é realizada uma ação com o pacote de acordo com a regra; caso contrário, (4) é feito um repasse do pacote ao controlador, que por sua vez (5) verifica se será necessário criar uma nova regra para esse pacote ou se ele deverá simplesmente ser descartado. O controlador então (6) informa ao elemento de rede o procedimento a ser adotado em relação ao pacote e se deverá ser adicionada uma nova regra para que o elemento de rede (7) encaminhe o pacote de acordo com essa nova regra ou descarte o pacote.

2.2 Orquestrador SDN

Um orquestrador SDN permite uma visão global dos recursos de rede, através da disponibilização de uma API para que um controlador ou grupo de controladores possam coordenar um número maior de nós de rede (SEZER et al., 2013). O OpenDaylight pode ser considerado como o principal orquestrador para redes SDN, e consiste de um projeto *open source* desenvolvido na linguagem de programação Java, mantido pela Linux Foundation e apoiado pelos principais fabricantes de dispositivos de rede, tais como IBM, Cisco, Juniper e VMWare (OPENDAYLIGHT, 2016).

A Figura 2.3 apresenta a arquitetura do controlador OpenDayligh, onde é possível perceber a presença de três camadas. Analisando de um visão *bottom-up*, a primeira delas apresenta um conjunto de protocolos e *plugins* suportados pelo OpenDayligh, tais como OpenFlow (versões 1.0 e 1.3), OVSDB (*Open vSwitch DataBase*, ou Banco de Dados vSwitch Aberto), NetConf,

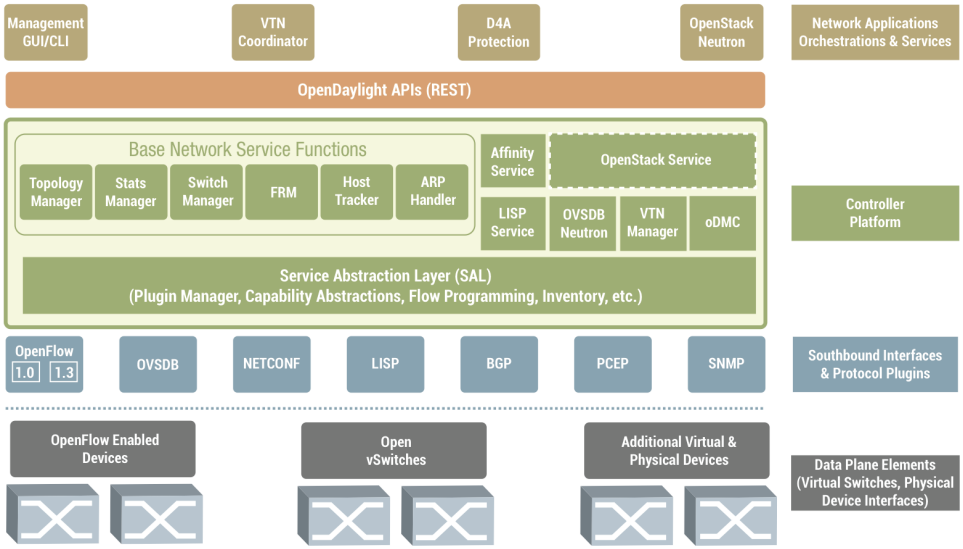


Figura 2.3 – Arquitetura do controlador OpenDaylight. Fonte: (OPENDAYLIGHT, 2016)

LISP (*Location Identifier Separation Protocol*, ou Protocolo de Localização e Separação de Identificador), BGP (*Border Gateway Protocol*, ou Protocolo de Gateway de Borda), PCEP (*Path Computation Element Protocol* ou Protocolo de Computação do Caminho do Elemento) e SNMP.

Na camada mais acima, é implementada toda a lógica de controle do OpenDayligh, composta por um conjunto de funções e serviços de rede, que podem ser acessados através de uma API (CISCO, 2014), na qual é fornecido um *Web Service*, contendo uma interface de acesso aos serviços através dos seguintes objetos remotos:

- ControllerManagerNorthbound - permite gerenciar o controlador, fornecendo funções para obter sua lista de propriedades (tais como o endereço MAC), alterar ou excluir uma existente;
- CustomPropertyNorthbound - permite a criação e utilização de propriedades customizáveis, que podem ser utilizadas pelo controlador. Alguns exemplos destas propriedades são: latência e largura de banda;
- FlowProgrammerNorthbound - objeto contendo as funções utilizadas para obter todas as regras OpenFlow da rede, ou por um determinado elemento (*switch*, roteador e etc). Além de permitir a criação de uma nova regra, além da alteração e exclusão de uma existente;
- HostTrackerNorthbound - disponibiliza funções para obter todos os *hosts* ativos (um host ativo é aquele que enviou ao menos uma mensagem na rede) ou inativos da rede, e também para localizar um determinado *host* pelo seu endereço MAC, informando em qual *switch* se encontra conectado;
- MonitorNorthbound - permite personalizar a forma como é feita a entrega do tráfego de rede, para algum dispositivo de monitoramento que esteja atuando na rede;

- P2PNorthbound - permite uma abstração dos enlaces existentes em um caminho entre dois *switches*, possibilitando a criação de uma regra OpenFlow entre eles, sem a necessidade de conhecer todos os *switches* do caminho;
- ResourceManagerNorthbound - permite adicionar ou remover um dispositivo de rede, através do seu endereço IP;
- RoutingServiceNorthbound - permite a visualização e alteração de rotas presentes em determinado dispositivo de rede;
- SecureChannelNorthbound - existe a possibilidade de adicionar uma segurança no acesso à API, através de um mecanismo de autenticação com login e senha. Esse objeto fornece funções que permitem o gerenciamento desta autenticação, assim como ativá-la ou desativá-la;
- SliceManagerNorthbound - permite o gerenciamento de todos os *slices* e todas as VLANs contidos da rede;
- StaticRoutingNorthbound - permite o gerenciamento de todas as rotas estáticas presentes na rede;
- StatisticsNorthbound - objeto que contendo funções que permitem acessar os dados estatísticos armazenados pelo protocolo OpenFlow, a nível de porta, regra, ou tabela de fluxos. Até a versão 1.3 do protocolo OpenFlow (suportada pelo OpenDaylight Hydrogen), tais dados consistiam no total de pacotes e total de *bytes*;
- SubnetsNorthbound - utilizado para gerenciar as sub-redes da rede;
- SwitchNorthbound - responsável por prover métodos utilizados para obter informações sobre cada *switch* ou roteador presente na rede. Tais informações consistem basicamente no seu ID (identificador único utilizado pelo controlador para identificar cada *switch* na rede), endereço MAC e o total de portas disponíveis;
- TifNorthbound - parecido com o P2PNorthbound, porém mais abrangente, pois abstrai não somente os enlaces de um determinado caminho entre dois nós, mas como toda a topologia da rede. Isso permite a criação de uma regra OpenFlow, sem a necessidade de conhecer toda a topologia da rede;
- TopologyNorthboundJAXRS - fornece todas as ligações existentes entre os *switches* e roteadores presentes na rede;
- UserManagerNorthbound - permite gerenciar os usuários que possuem permissão de acessar a API, dando a possibilidade de alterar usuários existentes ou criar novos.

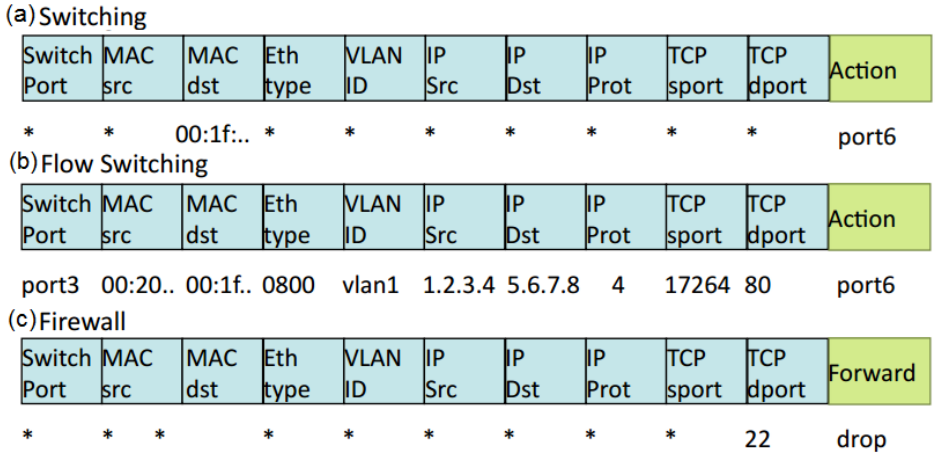


Figura 2.4 – Comutação de pacotes (a), comutação de fluxos (b) e filtro de pacotes (c). Fonte: (KATHI, 2014)

Já na camada mais acima, se encontram as aplicações e serviços de rede adicionais, que podem ser desenvolvidos através de uma interação com a API do OpenDayligh. Tais como aplicações e serviços de monitoramento e gerenciamento.

2.3 Elementos de Rede

Os elementos de rede podem ser tanto físicos quanto virtuais (como *switch* e roteador) e são responsáveis por realizar o encaminhamento dos pacotes na rede. Cada elemento de rede provê uma SBAPI para poder se comunicar com o controlador e receber as instruções que alimentam sua tabela de fluxos. Cada entrada da tabela identifica uma regra, que por sua vez identifica um fluxo, e determina a ação que deve ser tomada para cada pacote entrante, de acordo os campos contidos em seu cabeçalho, como Ethernet, IP, TCP, etc.

A Figura 2.4 apresenta três exemplos de regras de fluxo, onde 2.4(a) representa uma comutação simples de pacote, informando que para determinado MAC de origem a ação a ser tomada é encaminhar o pacote para a porta 6, em 2.4(b) mostra-se um exemplo de comutação de fluxo, sendo utilizados vários campos do cabeçalho do pacote, como VLAN ID, IP, TCP e MAC, tendo como ação encaminhar o pacote para a porta 6, e por último em 2.4(c) é implementado um filtro de pacotes que determina que todo pacote cuja porta de destino TCP seja 22 deve ser descartado.

2.4 Protocolo OpenFlow

O protocolo OpenFlow foi desenvolvido na Universidade Stanford com a finalidade de tornar os *switches* programáveis, o que permitiria aos pesquisadores testar novos protocolos nas redes atualmente em uso (MCKEOWN et al., 2008).

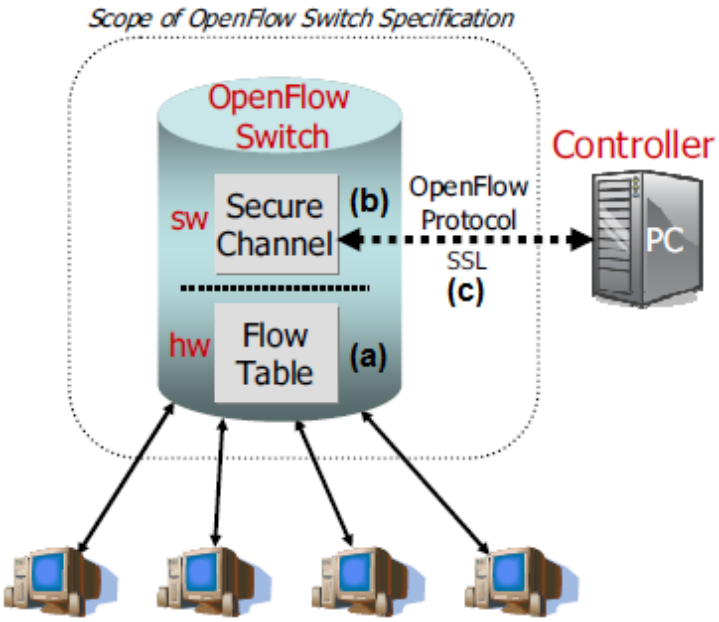


Figura 2.5 – Arquitetura do OpenFlow. Tabela de fluxos (a), canal seguro de comunicação (b) e a interface do protocolo OpenFlow (c). Fonte: (MCKEOWN et al., 2008)

Segundo o mesmo autor, o OpenFlow constitui o protocolo implementado em *switches* e roteadores que possibilita que estes recebam do controlador da rede as tabelas de fluxos. Nesse contexto, os elementos de rede podem ser de dois tipos: (i) *switches* OpenFlow dedicados e (ii) *switches*/roteadores com suporte a OpenFlow (ou híbridos). Enquanto os primeiros não realizam processamento nas camadas 2 e 3, os últimos possuem suporte a OpenFlow como uma característica adicional, podendo ser utilizada ou não.

A Figura 2.5 mostra os componentes de um *switch* OpenFlow, a saber: 2.5(a) uma tabela de fluxos, a qual possui uma ação associada a cada entrada, 2.5(b) um canal seguro (SSL) para transmissão de pacotes e instruções entre o *switch* e o controlador e 2.5(c) o protocolo OpenFlow, o qual constitui uma interface padronizada e aberta que permite a programação de suas tabelas a partir de um controlador central (dispensando a necessidade de programação in loco do equipamento). O cerne do funcionamento de um *switch* Openflow são as ações associadas aos fluxos de entrada, ambos presentes na tabela de fluxos interna do *switch*.

A Figura 2.6 apresenta os campos constituintes de uma entrada dessa tabela, quais sejam: 2.6(a) um cabeçalho para identificação dos pacotes, 2.6(b) uma ação que define como o pacote deve ser processado e 2.6(c) estatísticas (que podem ser utilizadas para remoção de fluxos inativos). Segundo McKeown et al. (2008) um *switch* OpenFlow dedicado pode implementar três modelos de ações:

1. Encaminhar o pacote por uma determinada porta a outro elemento de rede (ou o destino final, caso seja o último *hop*). Este é o comportamento padrão para a maioria dos fluxos;

2. Encapsular o pacote e enviar para o controlador. Isso acontece normalmente quando do

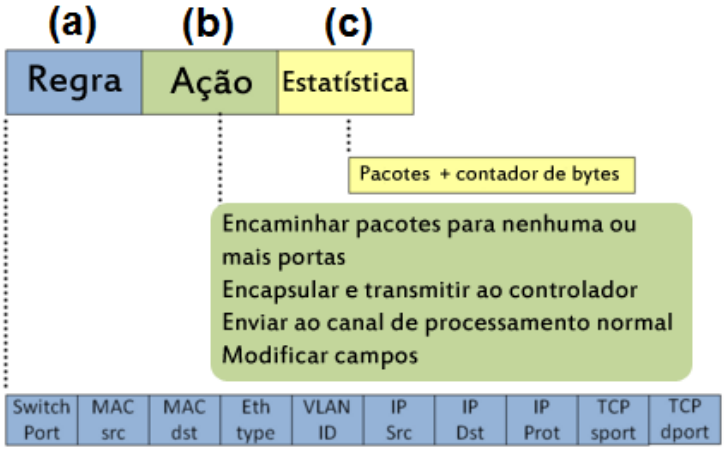


Figura 2.6 – Protocolo OpenFlow. Regra utilizada para criar um fluxo (a), ação que deve ser tomada para cada fluxo (b) e informações estatísticas que podem auxiliar o controlador (c). Fonte: (KATHI, 2014)

recebimento do primeiro pacote de um fluxo ainda não definido, mas pode ocorrer em outras situações, como em testes de protocolos;

3. Descartar o pacote recebido. Consiste em uma medida de segurança para evitar ataques conhecidos (por exemplo, DoS), pacotes *broadcast* de *discovery* de clientes, etc.

Para *switches*/roteadores com suporte a OpenFlow, ainda há mais uma ação possível:

1. Encaminhar o pacote para o pipeline normal de processamento. Uma vez que esses equipamentos podem ser utilizados para testes de novos protocolos enquanto em operação tradicional, pacotes não OpenFlow são encaminhados para o processamento normal.

Atualmente, a especificação do protocolo OpenFlow é mantida pela ONF, uma entidade aberta constituída por pesquisadores de universidades, administradores de rede, agências do governo e interessados em geral, desde que não sejam empregados de companhias fabricantes ou fornecedoras de *switches*, roteadores ou *wireless access points* (pontos de acesso sem fio). A maioria dos equipamentos lançados atualmente já possuem suporte a OpenFlow, inclusive vSwitches, como os presentes nos ambientes de virtualização Xen e KVM (VSWITCH, 2014).

Este capítulo introduziu o paradigma SDN apresentando os seus principais conceitos, bem como os principais componentes da sua arquitetura, com o objetivo de oferecer um referencial teórico, sobre o paradigma no qual foi desenvolvido o serviço de monitoramento de tráfego aqui proposto.

3 Monitoramento de Tráfego

Segundo Adrichem et al. (2014), o monitoramento de tráfego é considerado como um requisito fundamental para o gerenciamento de rede, principalmente quando se pretende utilizar mecanismos de QoS e engenharia de tráfego. Para (SLOMAN, 1994), o gerenciamento de um sistema consiste em supervisionar e controlar o seu funcionamento, com o objetivo de garantir que ele seja capaz de satisfazer determinados requisitos. Já para Verma e Verma (2009), gerenciar um sistema consiste na obtenção e consolidação de informações pertinentes ao estado e as configurações atuais dos elementos que o compõe. Desse ponto de vista, o monitoramento de tráfego de uma rede consiste em examinar periodicamente cada dispositivo presente na mesma, com o objetivo de apresentar informações tanto sobre o estado de cada um, quanto o estado de toda a rede ao seu administrador.

Para (LI; PAN, 2013), o monitoramento de tráfego pode ser utilizado na coleta de informações sobre o fluxo de dados em determinados pontos da rede, o que auxilia na aplicação de balanceamento de carga em *data centers*, por exemplo. Em (PHEMIUS; BOUET, 2013) o monitoramento de tráfego é utilizado para verificar a latência de determinados links da rede, sendo esta uma importante métrica para as aplicações de rede sensíveis ao atraso. Para (GIOTIS et al., 2014), o monitoramento de tráfego pode ser utilizado na identificação de padrões de ataques na rede em tempo real, permitindo que os ataques sejam mitigados antes de causarem algum problema à rede.

3.1 Fundamentação Teórica

No cenário que precede o proposto pelas redes SDN, o monitoramento de tráfego se mostra como algo crucial para as operações de gerenciamento, sendo posicionado dentro de uma sequência de três operações que devem ser executadas na rede (LEE; LEVANTI; KIM, 2014), são elas:

1. Projeto - tem como objetivo projetar mudanças o comportamento esperado e possíveis mudanças de configuração da rede, de acordo com requisitos e necessidades específicas. Com isso, é possível compreender as configurações existentes na rede, e mapeá-las com o comportamento esperado;

2. Implantação - operação que tem como objetivo implantar de forma segura, as configurações mapeadas na operação de projeto, garantindo que todas sejam implantadas nos dispositivos de rede, ou, em caso de algum problema, fazer com que a rede volte para último estado válido (*roll-back*). Além disso, a operação de implantação é responsável por garantir que os dispositivos de rede estejam com o seu *software* atualizado;

Tabela 3.1 – Comparação entre alguns protocolos de monitoramento e gerenciamento. Fonte: adaptada de (VERMA; VERMA, 2009)

	CLI	SNMP	CORBA	WBEM	WS	NetConf
Domínio Primário	Todos os Dispositivos	Rede	Servidores e Aplicações	Servidores e Aplicações	Servidores	Rede
Suporta Monitoramento	Sim	Sim	Sim	Sim	Sim	Não
Suporta Configuração	Sim	Não	Sim	Não	Sim	Sim
Segurança	Senha	Somente na versão 3	Sim	Sim, segurança HTTP	Sim, a ser definida	Sim
Restrições de Dados	Nenhuma	MIB	Nenhuma	CIM	Nenhuma	Nenhuma
Travessia pelo *firewall*	Fácil	Viável mas insegura	Pobre	Boa	Boa	Boa

3. Monitoramento - consiste em monitorar e medir o comportamento da rede, com o objetivo de identificar e resolver problemas, além de permitir a identificação de padrões de utilização da rede. Também permite verificar a exatidão de mudanças de configuração no dispositivos de rede.

Nesse processo de gerenciamento, são utilizados dois tipos de fontes de dados:

1. Medições - apresentam o comportamento atual da rede, as medições incluem pacotes coletados em diferentes pontos, assim como informações sobre determinado dispositivo de rede, que vão desde a carga de trabalho da CPU até entradas na tabela de repasse de um roteador, por exemplo;

2. Configurações - consistem de um conjunto de comando e parâmetros específicos para cada dispositivo da rede, que especificam qual deverá ser o comportamento esperado do dispositivo, bem como os protocolos que serão suportados pelo mesmo.

As medições mostram o comportamento atual da rede, ou seja, o seu estado, e podem incluir pacotes coletados em diferentes pontos da rede, assim como informações especificas de determinado dispositivo de rede, como a carga na sua CPU, por exemplo. Já as configurações de um dispositivo de rede, consistem em um conjunto de comados e parâmetros específicos,

utilizados para determinar como ele deve operar e quais protocolos devem estar em execução, por exemplo, além de incluírem informações sobre as conectividades física e lógica, com os demais dispositivos. Segundo Verma e Verma (2009), tanto as medições quanto as configurações, podem ser feitas através dos seguintes recursos e protocolos:

- CLI - consiste de uma interface de linha de comandos, que permite a entrada e saída de dados de um determinado dispositivo de rede (ou seja, *switch*, roteador e etc). Um exemplo de entrada, seria um conjunto de comandos utilizados para configurar o dispositivo. Enquanto que a saída, poderia ser composta por contadores, estatísticas ou indicadores de status do dispositivo. Uma maneira de executar as instruções em um dispositivo de rede, seria através de uma sessão *telnet*. Outra forma seria através de um login remoto, permitindo a execução das instruções, e obtendo-se o resultado da execução. Cada dispositivo de rede possui um determinado conjunto de instruções, que pode variar de acordo com o seu fabricante, por isso, ao utilizar este mecanismo para configurar ou gerenciar a rede, é necessário conhecer o conjunto de instruções de cada dispositivo presente na mesma;

- SNMP - protocolo de gerenciamento definido pelo IETF (*Internet Engineering Task Force*, ou Força Tarefa de Engenharia da Internet), amplamente utilizado para monitorar os dispositivos de rede. O SNMP atua na camada de aplicação do modelo OSI, e suporta os protocolos UDP e TCP na camada de transporte. Diferentemente do CLI, que apenas exibe os dados de monitoramento do dispositivo de rede, o SNMP os organiza de maneira hierárquica, indexando cada um deles em bases de informação, as chamadas MIBs (Management Information Base, ou Base de Informação de Gerenciamento). Isso acaba facilitando a busca por informações sobre o monitoramento da rede, além de permitir a criação de soluções capazes de ler os dados contidos nas MIBs de forma automatizada. Segundo Verma e Verma (2009) o protocolo SNMP possui três versões, SNMPv1, SNMPv2 e SNMPv3;

- CORBA - enquanto o SNMP trabalha com MIBs, o protocolo CORBA foi desenvolvido com a capacidade de trabalhar com objetos de software distribuídos. Um objeto de software distribuído, consiste de uma entidade de software que foi desenvolvida em uma linguagem de programação orientada a objetos, e que pode ser acessado através de uma interface bem definida. No caso do CORBA, a interface responsável por definir os objetos chama-se IDL (*Interface Definition Language*, ou Linguagem de Definição de Interface). Definida a interface dos objetos, é feita a sua publicação no ORB (*Object Repository Broker*, ou Barramento de Repositório de Objeto), permitindo que eles possam ser utilizados por outros softwares. Do ponto de vista do monitoramento da rede, um conjunto de objetos de software distribuídos podem ser implementados nos dispositivos de rede, podendo ser utilizados para a entrada de novas configurações, ou a coleta de informações;

- WBEM - devido ao crescimento da Internet, o protocolo HTTP passou a ser suportado em quase todos os tipos de dispositivos, fazendo com que a maioria dos *firewalls* sejam configurados para permitir mensagens HTTP. O protocolo WBEM faz uso do HTTP, com o objetivo de prover uma comunicação com os dispositivos de rede. Tanto as configurações quanto a coleta de dados de monitoramento, são codificadas através de um arquivo no formato CIM-XML, que consiste em um arquivo XML que utiliza o padrão de representação CIM (*Common Information Model*, ou Modelo de Informação Comum). Ao utilizar o protocolo WBEM, cada dispositivo de rede que será monitorado deverá possuir um servidor HTTP, responsável por atender as requisições CIM-XML e enviar as respostas. Isso permite que um software de monitoramento possa implementar um cliente que se comunique com o servidor e obtenha informações sobre os dispositivos de rede, ou envie novas configurações;

- *Web Service* - semelhantemente ao CORBA, permite a execução de funções remotamente através de uma interface, que poder ser definida através de arquivos XML no padrão WSDL (*Web Service Definition Language*, ou Linguagem de Definição de Serviço Web), via XML sem utilizar o WSDL, ou via JSON. Interfaces implementadas no formato WSDL, geralmente são acessadas através do protocolo SOAP (*Simple Object Access Protocol*, ou Protocolo Simples de Acesso a Objetos) utilizando o HTTP. Enquanto que as interfaces implementadas em JSON ou via XML sem WSDL são acessadas via REST, também através do protocolo HTTP. Assim como o protocolo WBEM, os *Web Services* podem ser utilizados para realizar a comunicação com os dispositivos de rede;

- NetConf - consiste de um conjunto de comandos de configuração baseados em XML, que podem ser utilizados através do protocolo SSH. Através do protocolo NetConf, é possível ler, alterar, ou até mesmo remover, as informações de configuração de um determinado dispositivo de rede. O NetConf é um protocolo padrão definido pelo IETF em 2006, com o objetivo de permitir uma configuração remota dos dispositivos de rede.

A Tabela 3.1 apresenta um resumo sobre os protocolos e recursos de gerenciamento apresentados, fazendo uma comparação entre eles com relação ao domínio primário em que atuam, se possuem suporte a monitoramento de tráfego e a configuração remota dos dispositivos de rede, assim como a forma com que cada um representa os dados, e a sua relação com um possível *firewall* que possa estar na rede.

Visto a importância do monitoramento no processo de gerenciamento da rede, é possível notar que a sua principal função, consiste na criação de um modelo do estado de funcionamento atual da rede, o qual pode ser utilizado para identificar problemas na rede e até mesmo na realização de análises e planejamentos que podem inclusive alterar o projeto da rede (LEE; LEVANTI; KIM, 2014). Esse modelo é composto por cinco camadas (ver a Figura 3.1):

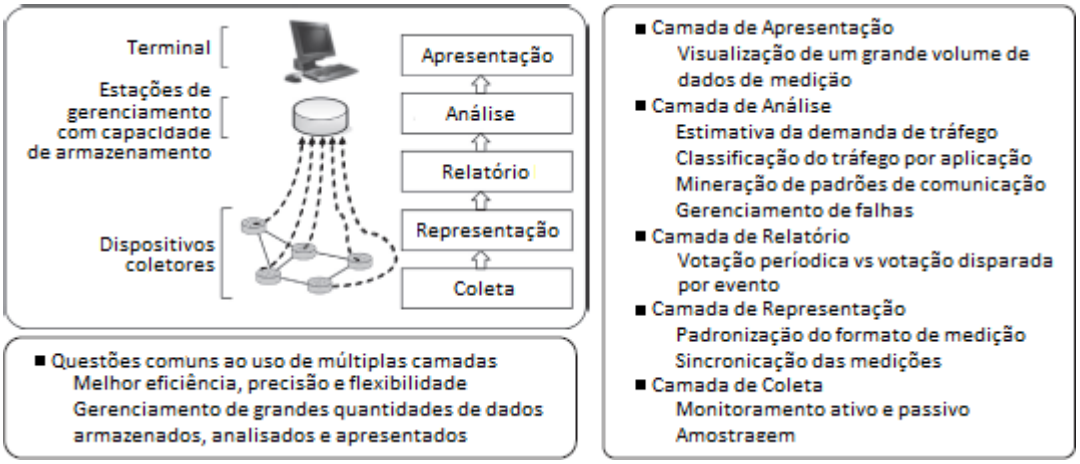


Figura 3.1 – Subclassificação das operações de monitoramento. Fonte: (LEE; LEVANTI; KIM, 2014)

1. Camada de Coleta - camada onde são coletados os dados brutos de medição da rede, ou seja, dados que ainda não receberam nenhum tipo de tratamento ou análise;

2. Camada de Representação - recebe os dados de medição coletados na camada de coleta, processa-os e os formata, com o objetivo de facilitar a sua interpretação;

3. Camada de Relatório - envia os dados de medição coletados e formatados para um conjunto de estações de gerenciamento, onde geralmente se encontram as camadas de análise e apresentação;

4. Camada de Análise - camada responsável por analisar e interpretar os dados de medições;

5. Camada de Apresentação - apresenta os dados de medição processados e analisados para os operadores de rede.

Os métodos de monitoramento de rede se encontram divididos em dois grupos, ativos e passivos (LEE; LEVANTI; KIM, 2014). Os métodos de monitoramento ativos injetam tráfego adicional à rede para medir o seu comportamento, como é o caso do *ping*, que faz uso de pacotes ICMP confiáveis para determinar o estado de uma conexão ponto a ponto, bem como o tempo de ida e volta. O problema desta abordagem, é que o tráfego adicional gerado pode influenciar nos resultados das medições da rede.

Já os métodos passivos, medem o tráfego da rede por meio de observação, sem a injeção de pacotes na rede, o que se mostra como a sua principal vantagem, pois evita que uma sobrecarga adicional seja adicionada, logo, não influencia no desempenho da rede. Porém, realizar um monitoramento passivo pode requerer grandes investimentos, devido à necessidade de instalar monitores de tráfego. Tanto os métodos ativos quanto os passivos são muito úteis para monitorar o tráfego da rede e coletar dados estatísticos, porém, é importante analisar qual utilizar levando em consideração os *trade-offs* existentes em cada um.

Em ambas as abordagens existe a necessidade de haver a figura de um coletor, que consiste em um elemento adicionado à rede com o objetivo de capturar os pacotes. O coletor pode ser implementado tanto via software quanto via hardware. Uma vez coletados os pacotes, o coletor os envia para o monitor, responsável por processá-los ou simplesmente exibi-los ao administrador da rede. Uma forma de implementar um coletor via hardware seria através de um tap (ONF, 2016).

Existem dois protocolos de monitoramento bastante utilizados, que fazem uso de coletores para capturar o tráfego da rede, o NetFlow (CLAISE, 2004) e o sFlow (PHAAL, 2004; SFLOW, 2015). Ambos trabalham com um esquema de *push* através da instalação de agentes em cada um dos elementos da rede, sendo responsáveis por capturar amostras de pacotes e enviá-las para um coletor, que consiste de um nó central da rede. Cada amostra, possui em seu conteúdo do cabeçalho uma determinada quantidade de pacotes, de acordo com a taxa de amostragem, permitindo uma estimativa do total de *bytes* transmitidos, assim como a taxa de transmissão. Por exemplo, caso seja utilizada uma taxa de amostragem igual a 100, o agente irá gerar uma amostra a cada 100 pacotes trafegados. A principal diferença entre os dois protocolos, é que o sFlow é aberto, enquanto que o NetFlow é de propriedade da Cisco.

3.2 Trabalhos Relacionados

Outra forma de monitorar o tráfego em uma rede, é através do protocolo OpenFlow, o qual permite que dados estatísticos sejam coletados dos elementos de rede (ADRICHEM et al., 2014). Tais dados consistem basicamente na quantidade de pacotes que passaram por determinada porta e o total de pacotes de um determinado fluxo, ou seja, os pacotes que se encaixarem em cada regra da tabela de fluxos (SUH et al., 2014). Para obter tais dados, é necessário que o controlador envie uma requisição ao elemento de rede, que por sua vez envia uma resposta, ambas as mensagens do protocolo OpenFlow (PHEMIUS; BOUET, 2013).

Atualmente existem vários estudos sobre o plano de controle em SDN (KREUTZ et al., 2015; GUEDES et al., 2012; SILVA et al., 2013), e alguns relacionados com o monitoramento do plano de dados. Porém, estes estudos exploram cenários bastante específicos, como o monitoramento de latência através de sondas (ou *probes*) para fazer a coleta de dados estatísticos presentes no protocolo OpenFlow em cada enlace da rede (PHEMIUS; BOUET, 2013), ou o monitoramento de transmissões TCP (SUH et al., 2014), ou ainda o monitoramento dos elementos de entrada e saída da rede (ADRICHEM et al., 2014).

Para Li e Pan (2013) uma coleta de dados estatísticos pode ser utilizada na aplicação de balanceamento de carga em *data centers*. Atualmente, a arquitetura de rede Fat-tree é uma das mais utilizadas em *data centers* para ser feito o balanceamento de carga na rede (LI; PAN, 2013). Porém, as abordagens apresentadas para a utilização da arquitetura Fat-tree, ainda não conseguem fazer um balanceamento de carga de forma totalmente satisfatória, pelo fato de

não apresentarem um mecanismo eficiente para a coleta de dados estatísticos dos elementos de rede. Para resolver este problema, é apresentada em (LI; PAN, 2013) a utilização do protocolo OpenFlow para coletar os dados estatísticos da rede dos *data centers*, e utilizá-los para fazer o balanceamento de carga em uma arquitetura de rede Fat-tree.

Segundo Phemius e Bouet (2013), uma coleta de dados estatísticos também pode ser utilizada para monitorar a latência dos links da rede, sendo esta uma métrica crucial a ser considerada com relação às aplicações sensíveis ao atraso, como é o caso das transmissões de vídeo via *stream*, transmissão VoIP e videoconferências. Phemius e Bouet (2013) propõem então a utilizados dados estatísticos (quantidade de pacotes e o total de *bytes*) do protocolo OpenFlow, para medir a latência experimentada em cada enlace. Para fazer tal medição, é feita a injeção de mensagens em um *switch* da malha de elementos de rede através do controlador, sendo recuperadas posteriormente em um *switch* adjacente. Dessa forma, a latência é calculada com base no tempo gasto para que as mensagens retornem ao controlador.

Em (GIOTIS et al., 2014), a coleta de dados estatísticos em uma SDN é utilizada para detectar padrões de ataques na rede em tempo real, bem como, impor regras para resolver os ataques através do protocolo OpenFlow. Para que isso ocorra, é proposta uma arquitetura com o objetivo de detectar e mitigar anomalias em redes SDN. Isso é feito através da coleta de dados estatísticos com o sFlow, que são repassados ao controlador, para que o mesmo execute um algoritmo que detecta padrões de ataque na rede, e envie novas regras para os elementos de rede, fazendo com que os ataques sejam mitigados.

Alguns trabalhos que abordam o monitoramento de tráfego em SDN, utilizam o protocolo sFlow para fazer a coleta de dados de tráfego da rede. Segundo (SUH et al., 2014), o sFlow atua somente no nível de granularidade das portas de cada elemento de rede, sendo proposta então uma modificação do protocolo. A qual consiste em inspecionar o número de sequência do TCP presente nos pacotes. Tal modificação apresentou de fato uma melhoria no funcionamento do sFlow, porém acabou restringindo o seu uso somente para transmissões TCP.

Um mecanismo de *push* também é utilizado em (ADRICHEM et al., 2014), o que permitiu uma análise fim-a-fim do atraso experimentado nos enlaces da rede. Também é utilizado um mecanismo de *polling* para coletar dados da rede, sendo medidas a largura de banda consumida por cada fluxo e a taxa de perda de pacotes.

Em (REZENDE et al., 2015) é apresentada uma plataforma de monitoramento de tráfego em SDN visando o provisionamento de QoS. Neste trabalho, são propostas duas abordagens para monitorar o tráfego, sendo uma baseada em amostragem utilizando o sFlow, e outra baseada no mecanismo de *polling*. Além disso, foram apresentadas fórmulas para o cálculo da taxa de transmissão e do atraso experimento por um fluxo fim-a-fim em uma rede SDN. Com isso, (REZENDE et al., 2015) provê um monitoramento de tráfego e permite ao operador (ou administrador da rede) alterar o caminho de um determinado fluxo, com base nas informações apresentadas sobre o tráfego da rede.

Além do QoS, também é possível analisar o QoE das aplicações em uma rede SDN, como proposto por (JARSCHEL et al., 2013), onde foram analisadas transmissões de vídeo via *stream* oriundas do YouTube. Fazendo o uso das informações mantidas pelos contadores previstos na especificação protocolo OpenFlow, (JARSCHEL et al., 2013) monitoraram o tráfego de dados, e aplicaram algumas técnicas para aplicar o balanceamento de carga visando melhorar o QoE da aplicação que estava sendo analisada, nesse caso o YouTube. Dentre as técnicas apresentadas, duas são independente de aplicação, sendo assim mais genéricas (Round-Robin e *Bandwidth-Based*), e uma, chamada Application-Aware, depende da aplicação, podendo ser aplicada em casos mais específicos.

Este capítulo apresentou o principais conceitos relacionados ao monitoramento de tráfego, tanto no âmbito das redes tradicionais, quanto das redes SDN, bem como as principais técnicas de monitoramento de tráfego utilizadas.

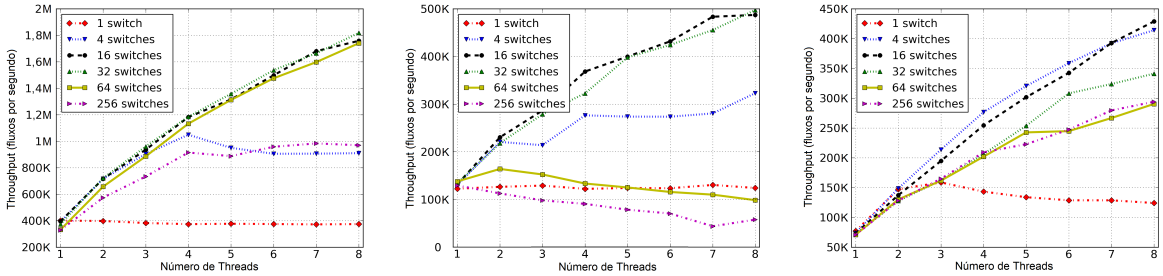
4 Desempenho de Controladores SDN

Segundo Tootoonchian et al. (2012), a utilização das redes SDN vem sido amplamente discutida, e se mostra como a solução mais adequada para vários tipos de cenários, podendo ser aplicada em redes domésticas, empresariais e até mesmo em *data centers*. Embora a adoção das redes SDN esteja sendo amplamente encorajada, o fato dos elementos de rede delegarem o seu plano de controle para um sistema externo, tem gerado algumas dúvidas e preocupações, tais como: (i) quão rápido um controlador pode atender as requisições feitas pelo plano de dados, ou seja, pelos elementos de rede? (ii) quantas requisições um controlador é capaz de atender por segundo? Devido a tais questionamentos, alguns estudos foram feitos visando avaliar o desempenho dos controladores SDN.

Um dos primeiros estudos realizados, avaliou o desempenho do primeiro controlador desenvolvido, o NOX (GUDE et al., 2008), e chegou a conclusão de que o mesmo é capaz de suportar a inicialização de até 30 mil novos fluxos por segundo, levando aproximadamente 10 ms para instalar cada fluxo (TAVAKOLI et al., 2009). O problema é que esse número ainda está muito longe do ideal, pois segundo Kandula et al. (2009) já naquele ano (2008) existiam servidores capazes de suportar até 100 mil fluxos por segundo, e para Benson, Akella e Maltz (2010) uma rede com 100 *switches* poderia apresentar uma taxa de até 10 milhões de fluxos por segundo. Além disso, os 10 ms gastos para instalar cada novo fluxo pode causar um atraso de 10% na maioria dos fluxos na rede.

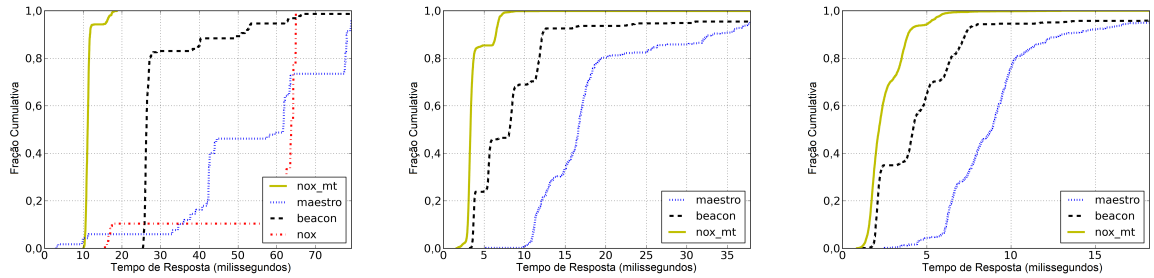
Essa diferença apresentada entre a demanda que pode ser apresentada à um controlador e o desempenho do controlador NOX, mostrou a necessidade de melhoria do mesmo, ou até mesmo a criação de um novo controlador mais eficiente, o que motivou o desenvolvimento do NOX-MT, uma versão do NOX com suporte a *multi-threaded* (TOOTOONCHIAN et al., 2012). Para avaliar tal controlador, foi proposta uma ferramenta para *benchmark* de controladores chamada cbench, que tem como objetivo emular um determinado número de *switches* OpenFlow, gerando requisições para o controlador avaliado, permitindo medir o seu desempenho através de critérios como *throughput* máximo, latência do *throughput*, além do tempo de resposta mínimo e tempo de resposta máximo. A avaliação do NOX-MT foi feita juntamente com o NOX e outros dois controladores, o Beacon (ERICKSON, 2013) e o Maestro (CAI, 2011).

A Figura 4.1 apresenta os resultados da avaliação proposta no que diz respeito ao *throughput* máximo, onde além da quantidade de *switches*, foi variada a quantidade de *threads*. Uma questão interessante sobre essa avaliação, é que não foram apresentados resultados para o NOX, aparentemente pelo fato do mesmo não suportar *multi-threaded*. Nesta avaliação, foi possível perceber que o NOX-MT apresentou um *throughput* maior do que os demais controladores, além de se mostrar mais estável com 16, 32 e 64 *switches*. Todos os controladores



(a) NOX-MT (b) Beacon (c) Maestro

Figura 4.1 – *Throughput* máximo para os controladores NOX-MT (a), Beacon (b) e (Maestro), com diferentes quantidades de *switches*. Fonte: adaptada de (TOOTOONCHIAN et al., 2012)



(a) Uma *thread* (b) Quatro *threads* (c) Oito *threads*

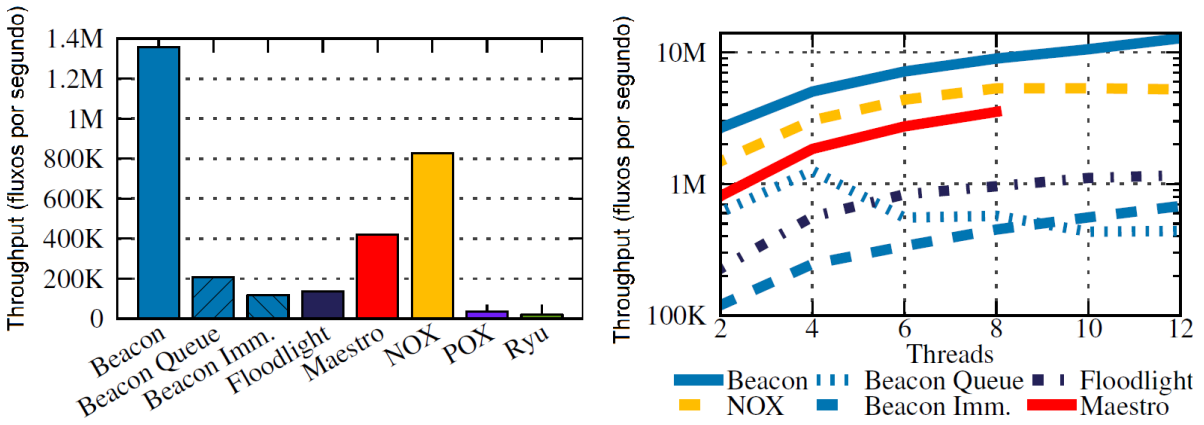
Figura 4.2 – Tempo de resposta dos controladores NOX, NOX-MT, Beacon e Maestro, com 32 *switches*, variando o número de *threads*. Fonte: adaptada de (TOOTOONCHIAN et al., 2012)

apresentaram uma perda de desempenho com 256 *switches*. Com relação ao tempo de resposta, o NOX-MT apresentou o menor valor, igual a 1,4 milissegundos durante a execução com 8 *threads*.

Por fim, (TOOTOONCHIAN et al., 2012) conclui que entre os controladores avaliados, o NOX-MT se mostrou mais eficiente, atendendo 1,6 milhões de requisições por segundo, com um tempo médio de resposta de 2 milissegundos. Além disso, a ferramenta de *benchmark* apresentada, chamada de cbench, consiste no primeiro passo para permitir a realização de experimentos, que possibilitem uma melhor compreensão de questões relacionadas ao desempenho dos controladores SDN.

Mais tarde, uma análise dos controladores SDN também é proposta em (ERICKSON, 2013), onde é feita uma comparação entre os mesmos, no que diz respeito ao *throughput* e a latência. Os controladores analisados foram: Beacon, Floodlight, Maestro, NOX, POX e Ryu, além de outras duas versões do Beacon, o Beacon Queue e o Beacon Immediate.

A Figura 4.3 (a) apresenta os resultados obtidos para os controladores com apenas uma *thread*, onde foi possível perceber um melhor desempenho do Beacon, seguido pelo NOX e pelo Maestro. Vale a pena observar que a versão do NOX utilizada suporta *multi-threaded*, como pode



(a) Uma *thread* (b) De duas a doze *threads*

Figura 4.3 – *Throughput* dos controladores Beacon, Beacon Queue, Beacon Imm., Floodlight, Maestro, NOX, POX e Ryu, variando o número de *threads*. Fonte: adaptada de (ERICKSON, 2013)

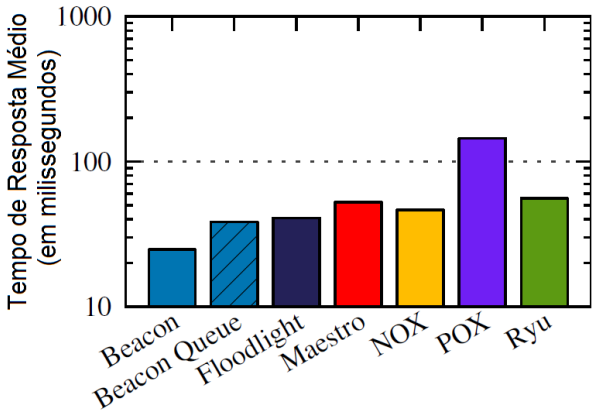


Figura 4.4 – Tempo de resposta médio dos controladores Beacon, Beacon Queue, Floodlight, Maestro, NOX, POX e Ryu. Fonte: adaptada de (ERICKSON, 2013)

ser visto na Figura 4.3 (b). Variando o número de *threads*, foi possível perceber que o Beacon permaneceu com o melhor desempenho, seguido pelo NOX e pelo Maestro, com o detalhe de que o Maestro suportou somente até 8 *threads*.

Com relação a latência, a Figura 4.4 apresenta o tempo de resposta médio dos controladores, onde foi possível perceber que o Beacon apresentou o menor tempo médio, seguido pelo Beacon Queue e pelo Floodlight, com o NOX ocupando a quarta posição. Por fim, é possível perceber uma certa similaridade nos resultados apresentados em (TOOTOONCHIAN et al., 2012) e (ERICKSON, 2013).

Shalimov et al. (2013) realizaram um levantamento dos controladores SDN disponíveis no ano de 2013, e segundos eles, os controladores elicitados assemelham-se aos sistemas operacionais utilizados por mainframes entre os anos 40 e 50. Assim como existia uma grande variedade de sistemas operacionais para os mainframes, cada um desenvolvido por um fabricante diferente, naquele ano existiam mais de 30 controladores diferentes desenvolvidos por fabricantes,

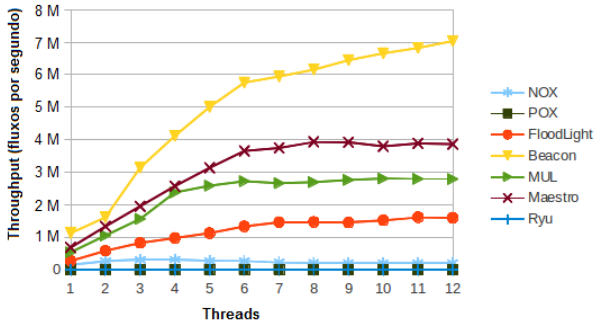


Figura 4.5 – *Throughput* dos controladores NOX, POX, Floodlight, Beacon, MuL, Maestro e Ryu variando o número de *threads*. Fonte: adaptada de (SHALIMOV et al., 2013)

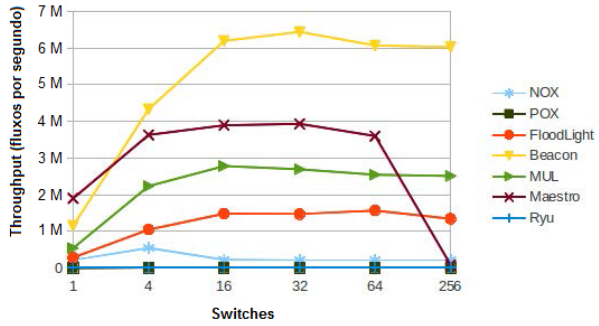


Figura 4.6 – *Throughput* dos controladores NOX, POX, Floodlight, Beacon, MuL, Maestro e Ryu variando o número de *switches*. Fonte: adaptada de (SHALIMOV et al., 2013)

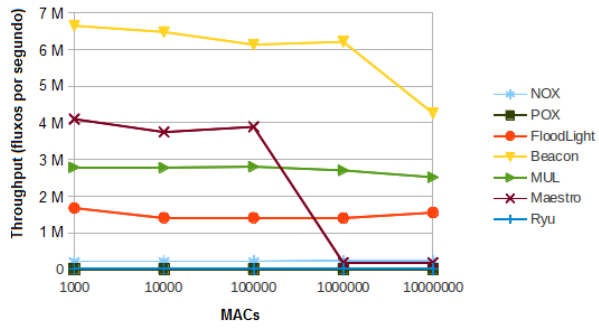


Figura 4.7 – *Throughput* dos controladores NOX, POX, Floodlight, Beacon, MuL, Maestro e Ryu variando o número de *MACs*. Fonte: adaptada de (SHALIMOV et al., 2013)

universidades e grupos de pesquisa (diferentes linguagens de programação e uso de *thread*), sendo os sete a seguir os mais utilizados:

1. NOX - controlador escrito na linguagem de programação C++ sem suporte a *multi-threaded* (GUDE et al., 2008);

2. POX - controlador escrito na linguagem de programação Phyton com suporte somente a *single-threaded*. É muito utilizado na criacão de protótipos de aplicacões de rede para pesquisas (POX, 2015);

Tabela 4.1 – Tempo de resposta mínimo (10^{-6} segundos por fluxo). Fonte: adaptada de (SHALIMOV et al., 2013)

Controlador	Tempo de Resposta
NOX	91,531
POX	323,443
Floodlight	75,484
Beacon	57,205
MuL	50,082
Maestro	129,321
Ryu	105,58

Tabela 4.2 – Resultados dos testes de segurança. Fonte: adaptada de (SHALIMOV et al., 2013)

Controlador	1	2	3	4	5
NOX					
POX					
Floodlight					
Beacon					
MuL					
Maestro					
Ryu					

3. Beacon - controlador escrito na linguagem de programação Java com suporte a *multi-threaded*. Desenvolvido em cima dos *frameworks* OSGI e Spring (ERICKSON, 2013);

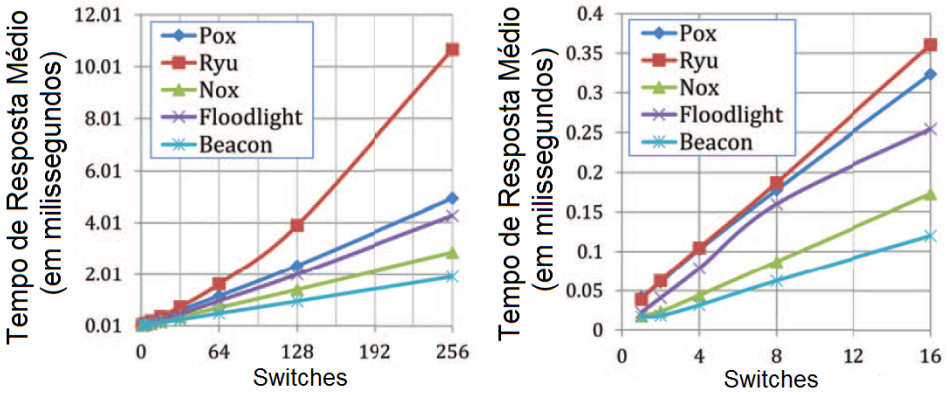
4. Floodlight - controlador escrito na linguagem de programação Java com suporte a *multi-threaded*. Desenvolvido sobre o *frameworks* Netty (FLOODLIGHT, 2015);

5. MuL - controlador escrito na linguagem de programação C com suporte a *multi-threaded* (MUL, 2015);

6. Maestro - controlador escrito na linguagem de programação Java com suporte a *multi-threaded* (CAI, 2011);

7. Ryu - controlador escrito na linguagem de programação Phyton (RYU, 2015).

Além das medidas de desempenho, tais como *throughput*, escalabilidade e tempo de resposta, em (SHALIMOV et al., 2013) são avaliadas a confiabilidade e a segurança dos controladores. Para que isso fosse possível, foi proposta uma ferramenta de *benchmark* chamada hcprobe. Foram avaliados os seguintes controladores: NOX, POX, Beacon, Floodlight, MuL, Maestro e Ryu.



(a) Overall (b) Zoom

Figura 4.8 – Tempo de resposta médio dos controladores POX, Ryu, NOX, Floodlight e Beacon variando o número de *switches*. Fonte: adaptada de (ZHAO; IANNONE; RIGUIDEL, 2015)

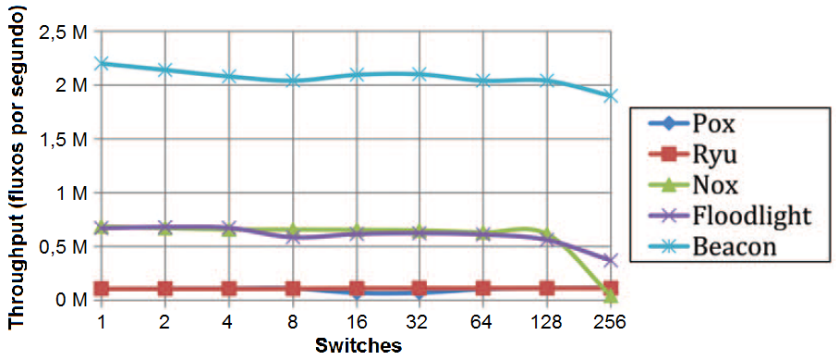
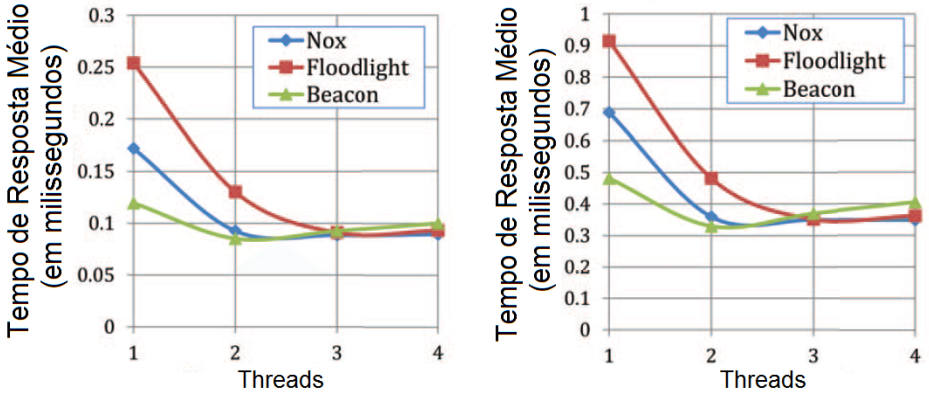


Figura 4.9 – *Throughput* dos controladores POX, Ryu, NOX, Floodlight e Beacon variando o número de *switches*. Fonte: adaptada de (ZHAO; IANNONE; RIGUIDEL, 2015)

Com relação ao *throughput* e a escalabilidade, os controladores POX e Ryu não suportam *multi-threaded*, fazendo com que não possuam boa escalabilidade. Já o Maestro apresenta uma boa escalabilidade suportando até 8 *threads*. O Floodlight apresenta uma melhora no desempenho ao variar o número de *threads*, até atingir 8, sem apresentar uma melhora significativa entre 9 e 12 *threads*. O Beacon apresentou o melhor *throughput*, com cerca de 7 milhões de fluxos por segundo. O resultados podem ser vistos na Figura 4.5.

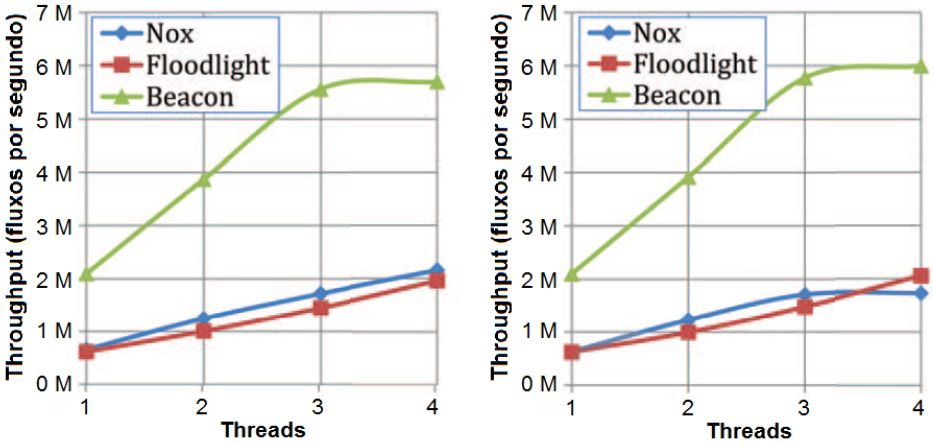
Variando-se o número de *switches*, foi possível perceber um melhor *throughput* com relação aos controladores que suportam *multi-threaded*. Entre os controladores, o Maestro apresentou um melhor resultado com uma quantidade menor de *switches*, apresentando uma queda no número de fluxos suportados a medida em que aumentou a quantidade de *switches*. Apesar de apresentar inicialmente um *throughput* menor que o Maestro, o Beacon se manteve como o controlador com o maior *throughput*, como mostra a Figura 4.6.

Com relação ao número de *hosts* (sendo um *host* representado por um endereço MAC único), foi possível perceber uma queda considerável no *throughput* dos controladores, o Maestro, por exemplo, apresentou essa queda com uma quantidade 10^6 *hosts* conectados. Já o Beacon,



(a) 16 *switches* (b) 64 *switches*

Figura 4.10 – Tempo de resposta médio dos controladores NOX, Floodlight e Beacon variando o número de *switches*, com o *Hyper-Threading* desativado. Fonte: adaptada de (ZHAO; IANNONE; RIGUIDEL, 2015)



(a) 16 *switches* (b) 64 *switches*

Figura 4.11 – *Throughput* dos controladores NOX, Floodlight e Beacon variando o número de *switches*, com o *Hyper-Threading* desativado. Fonte: adaptada de (ZHAO; IANNONE; RIGUIDEL, 2015)

apresentou somente quando a quantidade de *hosts* aumentou para 10^7 (ver a Figura 4.7).

A Tabela 4.1 apresenta o tempo de resposta médio dos controladores NOX, POX, Floodlight, Beacon, MuL, Maestro e Ryu, com 10^5 *hosts* conectados, onde é possível perceber um menor valor para os controladores NOX, Beacon e MuL.

Sobre a confiabilidade dos controladores, os experimentos apresentaram que a maioria suportou a carga experimentada, com exceção do MuL e do Maestro, que começaram a descartar pacotes após algum tempo em execução. O MuL descartou 660.271.177 mensagens e fechou 214 conexões, enquanto que o Maestro não fechou nenhuma conexão, porém descartou 463.012.511 mensagens.

Os experimentos que visaram avaliar a segurança dos controladores, foram realizados

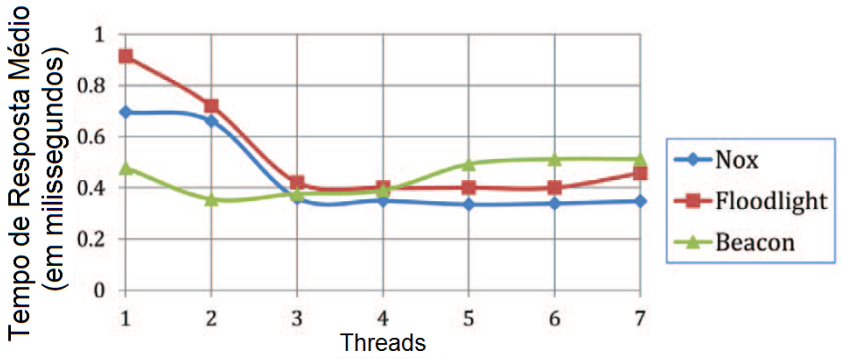


Figura 4.12 – Tempo de resposta médio dos controladores NOX, Floodlight e Beacon variando o número de *threads*, com o *Hyper-Threading* ativado. Fonte: adaptada de (ZHAO; IANNONE; RIGUIDEL, 2015)

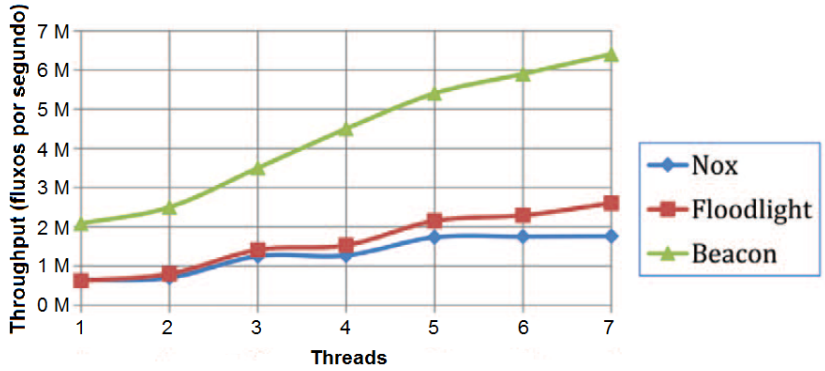


Figura 4.13 – *Throughput* dos controladores NOX, Floodlight e Beacon variando o número de *threads*, com o *Hyper-Threading* ativado. Fonte: adaptada de (ZHAO; IANNONE; RIGUIDEL, 2015)

através do envio de mensagens malformadas, com alterações feitas no cabeçalho (tamanho incorreto da mensagem (1), versão inválida do OpenFlow (2) e tipo da mensagem OpenFlow incorreta (3)) e no corpo (mensagem do tipo PacketIn alterada (4) e status da porta alterado (5)). As alterações nas mensagens foram feitas através da ferramenta hcprobe. A Tabela 4.2 apresenta os resultados sobre como os controladores agiram ao receberem os cinco tipos de mensagens malformadas. A cor vermelha indica que o controlador parou de funcionar, a cor laranja que o controlador fechou a conexão, e ver que passou no teste. A cor amarela indica que o controlador não parou de funcionar e nem fechou a conexão, porém não identificou as mensagens malformadas, podendo indicar uma falha de segurança.

Por fim, em (SHALIMOV et al., 2013) conclui-se que o controlador Beacon apresentou um melhor *throughput*, atendendo cerca de 7 milhões de fluxos por segundo, e maior escalabilidade, atuando com até 1 *threads*. Com relação à confiabilidade, os experimentos demonstraram que a maioria dos controladores é capaz de atuar sem sofrer quedas, com exceção do MuL e do Maestro. Sobre a segurança, foi possível perceber que ainda existem falhas nos controladores que precisam ser corrigidas, com o objetivo de evitar um possível ataque no futuro.

Além dos controladores citados anteriormente, existem os chamados orquestradores, que

são controladores que atuam de forma distribuída na rede (ZHAO; IANNONE; RIGUIDEL, 2015). Um exemplo deles é o OpenDayLight (OPENDAYLIGHT, 2016), que consiste no maior projeto *open source* colaborativo mantido pela Linux Foundation, com o objetivo de acelerar a adoção do paradigma SDN e criar uma base sólida para NFV (*Network Function Virtualization*, ou Virtualização da Função de Rede).

Em Zhao, Iannone e Riguidel (2015), também é proposta uma avaliação de alguns controladores, tanto centralizados quanto distribuídos, na qual é utilizado um recurso dos processadores Intel, chamado *Hyper-Threading*, que ao ser ativado melhora a execução das operações em paralelo.

A Figura 4.8 apresenta os resultados obtidos com relação ao tempo de resposta dos controladores centralizados, variando-se o número de *switches*, onde é possível perceber que tanto de uma visão macro (Figura 4.8 (a)), quanto de uma visão mais detalhada (Figura 4.8 (b)), é possível perceber um aumento no tempo de resposta a medida em que aumenta o número de *switches*. O mesmo comportamento também pode ser visto com relação ao *throughput* (ver a Figura 4.9), apresentando uma queda de desempenho com uma quantidade de *switches* maior ou igual a 128.

Variando-se o número de *threads* com o *Hyper-Threading* desativado, tanto com 16 *switches* (ver a Figura 4.10 (a)), quanto com 64 *switches* (ver a Figura 4.10 (b)), foi possível perceber uma queda no tempo de resposta com duas *threads*, e um certo aumento com quatro *threads*. Com relação ao *throughput*, houve um crescimento no número de fluxos atendidos, com uma aparente estabilidade ao atingir quatro *threads*, com 16 e com 64 *switches* (ver a Figura 4.11).

Ao ativar o *Hyper-Threading*, e variando-se o número de *threads* até 7, foi possível notar um comportamento parecido com relação ao tempo de resposta (ver a Figura 4.12), porém, foi possível perceber um aumento significativo no *throughput*, principalmente com relação ao Beacon, como mostra a Figura 4.13.

Por fim, Zhao, Iannone e Riguidel (2015) concluem que entre os controladores, o Beacon apresentou o melhor desempenho, e que o fato de não apresentarem uma comparação com o orquestrador OpenDayLight, se dá pela razão dele possuir uma complexidade muito superior a dos controladores, devido à sua natureza de atuar de forma distribuída na rede, exigindo um quantidade maior de recursos computacionais, e acarretando em uma sobrecarga maior, podendo alterar de forma negativa sua avaliação de desempenho.

A Tabela 4.3 apresenta um resumo sobre os principais controladores SDN, com relação a linguagem de programação utilizada para desenvolver cada um deles, as versões do protocolo OpenFlow que cada um suporta e o seu modo de atuação na rede, centralizado ou distribuído. Também é apresentada a fonte de cada controlador.

Neste capítulo, foram apresentados alguns estudos realizados com o objetivo de avaliar

Tabela 4.3 – Principais controladores SDN elicitados. Fonte: Criada pelo autor

	Controlador	Versão OpenFlow	Fonte
Centralizado			
	NOX (C++)	1.0 e 1.3	Gude et al. (2008)
	NOX-MT (C++)	1.0 e 1.3	Tootoonchian et al. (2012)
	POX (Phyton)	1.0	POX (2015)
	Beacon (Java)	1.0	Erickson (2013)
	Floodlight (Java)	1.0	Floodlight (2015)
	MuL (C)	1.0 até 1.4	MuL (2015)
	Maestro (Java)	1.0	Cai (2011)
	Ryu (Phyton)	1.0 até 1.4	Ryu (2015)
Distribuído			
	OpenDayLight (Java)	1.0 e 1.3	OpenDaylight (2016)

o desempenho de controladores SDN, sendo levados em consideração o *throughput*, a escalabilidade, o tempo de resposta, a confiabilidade e a segurança, como medidas de desempenho. Levando a conclusão de que houve uma evolução nos controladores SDN desde o NOX (primeiro controlador desenvolvido), e que a busca por um melhor desempenho levou ao desenvolvimento de controladores com suporte a *multi-threaded* e a criação de controladores distribuídos, como o OpenDayLight.

5 Serviço de Monitoramento para SDN

Este trabalho apresenta como objetivo geral, o desenvolvimento de um mecanismo que permita a realização do processo de monitoramento de tráfego em redes SDN. Visando alcançar esse objetivo, foi desenvolvido um serviço de monitoramento chamado SDNMonitor, que permite a realização do monitoramento de tráfego em redes SDN, em alguns níveis de granularidade.

O serviço de monitoramento SDNMonitor visa monitorar o estado da rede através de um mecanismo de *polling* utilizando os dados estatísticos previstos na especificação do protocolo OpenFlow (FOUNDATION, 2014), sem a necessidade de injetar mensagens na rede, como realizado em (PHEMIUS; BOUET, 2013). Outra característica, é o monitoramento fim-a-fim dos fluxos que utilizam tanto o TCP quanto o UDP, algo que não foi possível nos trabalhos (SFLOW, 2015; PHAAL, 2004; SUH et al., 2014). Além disso, o serviço aqui proposto se beneficia da principal característica do paradigma SDN, que é justamente prover uma visão global da rede, algo que não foi bem explorado em (ADRICHEM et al., 2014).

Com o objetivo de facilitar o gerenciamento da rede, além do serviço de monitoramento, foi desenvolvida uma aplicação Web, contendo algumas interfaces que permitem a realização de algumas operações. Adicionalmente, foi desenvolvido um serviço de balanceamento de carga, que permite realizar alterações na rede sem a necessidade de parar o seu funcionamento.

Para que o serviço de monitoramento fosse capaz de interagir com a rede, foi necessário realizar a escolha de um controlador, sendo optado então pelo OpenDayLight, um controlador SDN mantido pela Linux Foundation (OPENDAYLIGHT, 2016). Além de suportar a versão 1.3 do protocolo OpenFlow, o OpenDayLight possui uma vasta documentação, e sua última atualização até o momento em que este trabalho está sendo escrito, foi realizada em maio de 2016.

O controlador OpenDayLight apresenta uma arquitetura onde é possível ver todos os seus principais componentes dispostos em algumas camadas. Umas delas é destinada para aplicações e serviços de rede, sendo feita a sua comunicação com a camada de controle através de uma API bem definida. Na Figura 5.1, é possível ver o serviço de monitoramento aqui proposto, chamado SDNMonitor, situado na arquitetura do controlador OpenDayLight. Também é possível ver uma visão geral do SDNMonitor, onde são apresentadas as funções da aplicação Web desenvolvida (descritas na Seção 5.3 deste capítulo), assim como os serviços desenvolvidos (descritos na Seção 5.2 deste capítulo). É possível perceber que a aplicação Web se encontra na Camada de Aplicações da arquitetura SDN, e os serviços na Camada de Controle.

A fim de facilitar a compreensão deste capítulo, a partir deste momento a palavra fluxo será utilizada para indicar uma ligação ponto a ponto entre dois *hosts* da rede, levando em consideração os IPs utilizados (origem e destino) e os *switches* por onde passa a ligação (ou

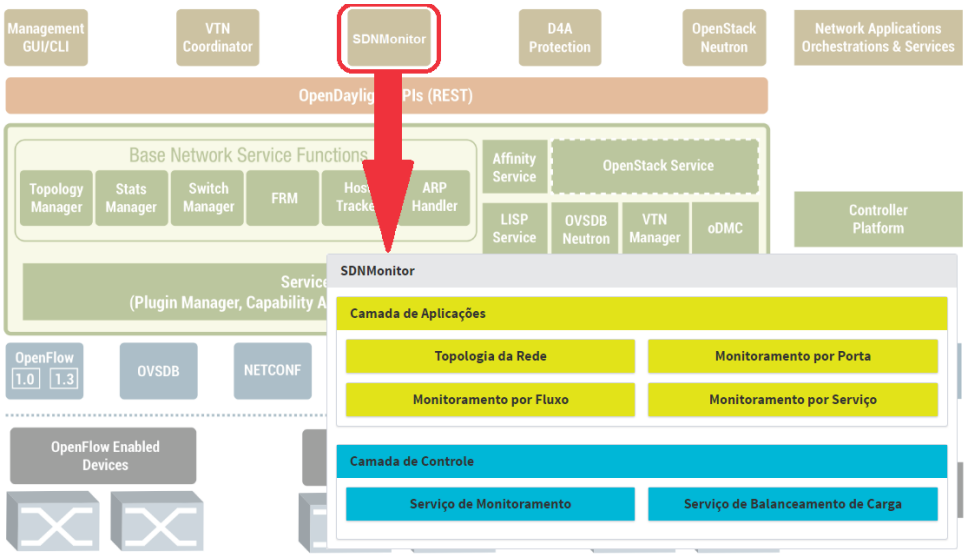


Figura 5.1 – Arquitetura do controlador OpenDaylight. Fonte: adaptada de (OPENDAYLIGHT, 2016)

seja, o caminho). Ao se referir à uma entrada na tabela de fluxos OpenFlow de um *switch*, será utilizado o termo regra do OpenFlow. E a palavra serviço, irá indicar ligação ponto a ponto entre dois *hosts* da rede levando em consideração os IPs utilizados (origem e destino), o protocolo da camada de transporte (UDP ou TCP), a porta utilizada (5001, 5201, 8080, 8081, 21 e etc) e os *switches* por onde passa a ligação.

5.1 Modelo de Dados

Com o objetivo de representar uma rede SDN, e permitir armazenar toda a sua estrutura, assim como os dados referentes ao monitoramento de tráfego, foi definido um modelo de dados (ver a Figura 5.2). Com base neste modelo, foi criado um banco de dados utilizando o SGBD MySQL, contendo as seguintes entidades e relacionamentos:

- Usuário (TB_USUARIO) - entidade responsável por representar um usuário da aplicação Web (descrita na Seção 5.3). Composta pelo seu identificador (um número inteiro gerado automaticamente pelo SGBD), pelo seu login e senha (utilizados para acessar a aplicação), pelo seu tipo (atualmente a aplicação só possui o tipo Administrador, porém pode haver a necessidade de criar outros tipos no futuro) e pelo seu nome;
- Nó da Rede (TB_NO_REDE) - com o objetivo de permitir uma análise mais genérica da rede, foi criada esta entidade, que tem como objetivo representar todo e qualquer dispositivo ativo na rede. Dessa forma, qualquer *host* ou *switch* precisa estar ligado à um nó, responsável por representá-lo. Entidade composta somente pelo seu identificador (um número inteiro gerado automaticamente pelo SGBD);

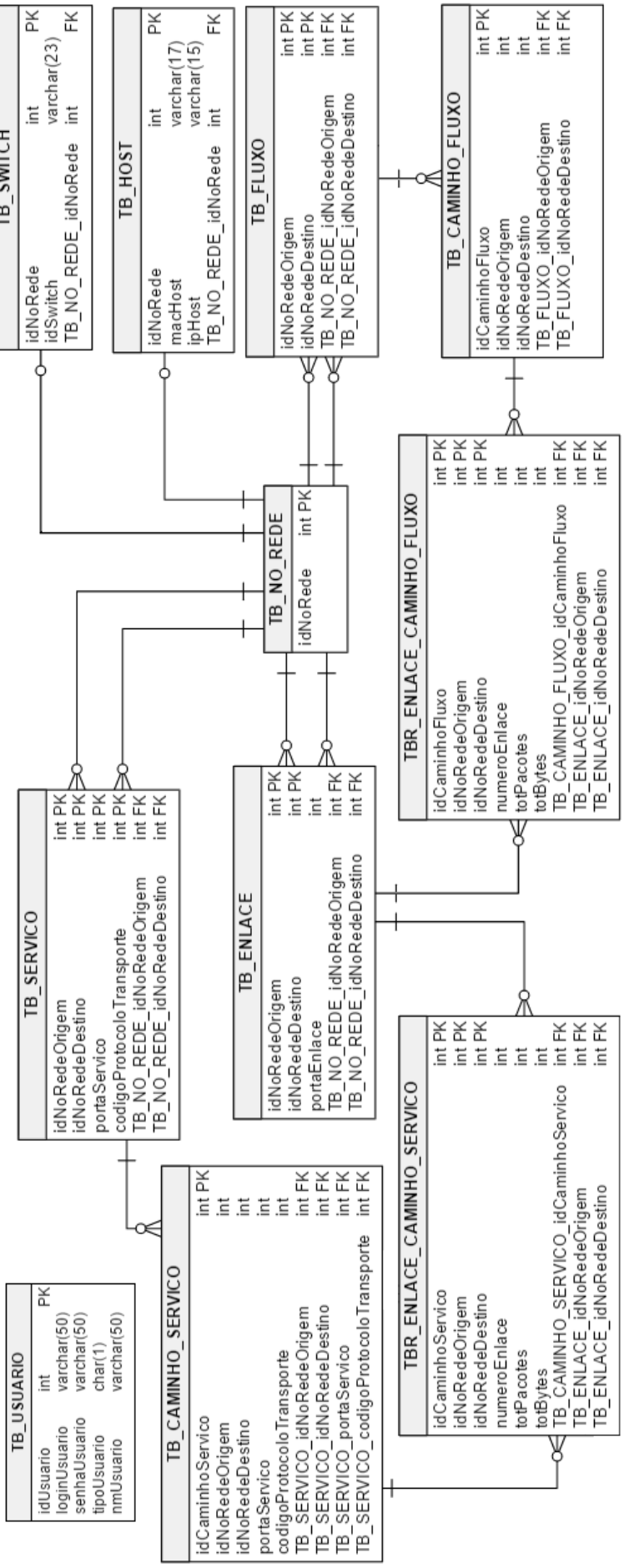


Figura 5.2 – DER do modelo de dados proposto para o serviço SDNMonitor. Fonte: Criada pelo autor

- Switch (TB_SWITCH) - entidade que representa um *switch* na rede, sendo composta pelo seu identificador (que consiste no identificador do nó da rede utilizado para representá-lo) e pelo seu ID na rede (obtido através do serviço de monitoramento descrito na Seção 5.2);

- Host (TB_HOST) - entidade responsável por representar um *host* na rede, sendo composta pelo seu identificador (que consiste no identificador do nó da rede utilizado para representá-lo), pelo seu endereço MAC e o seu endereço IP;

- Enlace (TB_ENLACE) - entidade que representa uma ligação física e direta entre dois nós da rede. Pelo fato de todos os *switches* e todos os *hosts* serem representados como nós, esta entidade consegue representar todos os enlaces presentes na rede, tanto entre *switches*, quanto entre um *switch* e um *host*. Composta pelo nó de origem, o nó de destino e o número da porta do enlace, que juntos formam o seu identificador;

- Fluxo (TB_FLUXO) - entidade criada para representar um fluxo entre dois nós da rede. Composta pelo nó de origem e o nó de destino, que juntos formam o seu identificador;

- Caminho do Fluxo (TB_CAMINHO_FLUXO) - um fluxo pode ter mais de uma opção de caminho, por isso, foi criada esta entidade com o objetivo de representar cada um deles. Composta pelo seu identificador (um número inteiro gerado automaticamente pelo SGBD) e pelo identificador do fluxo (nó de origem e nó de destino) ao qual o caminho pertence;

- Enlace no Caminho do Fluxo (TBR_ENLACE_CAMINHO_FLUXO) - cada caminho de um fluxo é composto por uma série de enlaces, sendo assim, esta entidade é responsável por representar o relacionamento entre um caminho (TB_CAMINHO_ FLUXO) e seus enlaces (TB_ENLACE). Composta pelo identificador do caminho (um número inteiro gerado automaticamente pelo SGBD) e o identificador do enlace (nó de origem, nó de destino e o número da porta do enlace), que juntos formam o seu identificador. Além do número do enlace (que identifica a ordem de cada enlace em cada caminho), o total de pacotes e o total de *bytes* coletados para cada enlace do caminho;

- Serviço (TB_SERVICO) - entidade criada para representar um serviço entre dois nós da rede. Composta pelo nó de origem, o nó de destino, a porta do serviço (5001, 5201, 8080, 8081, 21 e etc) e o código do protocolo da camada de transporte (descrito na Seção 5.2), que juntos formam o seu identificador;

- Caminho do Serviço (TB_CAMINHO_SERVICO) - um serviço pode ter mais de uma opção de caminho, por isso, foi criada esta entidade com o objetivo de representar cada um deles. Composta pelo seu identificador (um número inteiro gerado automaticamente pelo SGBD) e pelo identificador do serviço (nó de origem, nó de destino, porta do serviço e código do protocolo da camada de transporte) ao qual o caminho pertence;

- Enlace no Caminho do Serviço (TBR_ENLACE_CAMINHO_SERVICO) - cada caminho de um serviço é composto por uma série de enlaces, sendo assim, esta entidade é responsável por representar o relacionamento entre um caminho (TB_CAMINHO_ SERVICO) e seus enlaces (TB_ENLACE). Composta pelo identificador do caminho (um número inteiro gerado automaticamente pelo SGBD) e o identificador do enlace (nó de origem, nó de destino e o número da porta do enlace), que juntos formam o seu identificador. Além do número do enlace (que identifica a ordem de cada enlace em cada caminho), o total de pacotes e o total de *bytes* coletados para cada enlace do caminho;

5.2 Camada de Controle

5.2.1 Serviço de Monitoramento

Para implementar o serviço de monitoramento foi utilizada a API do Opendaylight versão Hydrogen. Além disso foi adotada a linguagem de programação Java. Para que a solução proposta fosse capaz de prover o monitoramento de tráfego em alguns níveis de granularidade (por porta do *switch*, por fluxo e por serviço), foi necessário obter as informações sobre toda a topologia da rede, sendo utilizado o modelo proposto para representar cada uma das entidades. Nesse processo, foram utilizados os seguintes objetos remotos providos pela API do OpenDaylight:

- TopologyNorthboundJAXRS - utilizado para obter dados da topologia da rede;
- SwitchNorthbound - utilizado para obter dados de cada *switch* da rede;
- HostTrackerNorthbound - utilizado para obter todos os *hosts* ativos na rede, bem como o *switch* ao qual cada um se encontra conectado.

5.2.1.1 Problema de Persistência dos Dados Estatísticos

Além desses objetos, foi utilizado o StatisticsNorthbound, com o objetivo de obter os dados estatísticos previstos pelo protocolo OpenFlow (total de pacotes e total de *bytes*). A coleta destes dados é feita de forma acumulativa, para cada regra presente na tabela de fluxos, de cada um dos *switches*, ou seja, é armazenado sempre o total de pacotes e o total de *bytes* que passaram por cada uma, desde o momento em que foram criadas. O problema disso, é que caso um determinado *switch* possa ser utilizado por mais de um caminho em um mesmo fluxo, não será possível saber a qual caminho pertencem os dados estatísticos. Por exemplo, caso exista uma regra para um determinado fluxo X (IP de origem 10.0.0.1 e IP de destino 10.0.0.2, por exemplo), que tenha como ação encaminhar os pacotes para a porta 2 do *switch*, caso seja feita uma mudança de caminho, que pode ser simplesmente uma alteração na ação, indicando que os pacotes devem ser encaminhados para a porta 3, toda a informação estatística obtida será perdida, pois para o protocolo OpenFlow uma nova regra foi criada.

Algoritmo 1 Manutenção da Topologia

```
1: procedure manterTopologiaRede()
2:     enlacesRede ← obterEnlacesRede()
3:     if sizeof(enlacesRede) > 0 then
4:         while i < sizeof(enlacesRede) do
5:             cadastrarSwitch(enlacesRede[i].getOrigem())
6:             cadastrarSwitch(enlacesRede[i].getDestino())
7:             cadastrarEnlaceSwSw(enlacesRede[i].getOrigem(),
   enlacesRede[i].getDestino())
8:             i ← i + 1
9:     else
10:        switchesRede ← obterSwitchesRede()
11:        while i < sizeof(switchesRede) do
12:            cadastrarSwitch(switchesRede[i])
13:            i ← i + 1
14:    hostsRede ← obterHostsRede()
15:    while i < sizeof(hostsRede) do
16:        cadastrarHost(hostsRede[i], hostsRede[i].getSwitch())
17:        i ← i + 1
```

Algoritmo 2 Definição dos Caminhos

```
1: procedure montarCaminhosHosts(switchInicioCaminho, switchFimCaminho,
   switchesCaminho)
2:     hostsInicioCaminho ← switchInicioCaminho.getHostsSwitch()
3:     if sizeof(hostsInicioCaminho) > 0 ∧ switchInicioCaminho.getId() ≠
   switchFimCaminho.getId() then
4:         hostsFimCaminho ← switchFimCaminho.getHostsSwitch()
5:         while i < hostsInicioCaminho do
6:             while j < hostsFimCaminho do
7:                 cadastrarCaminhoHosts(hostsInicioCaminho[i], hostsFimCaminho[j],
   switchesCaminho)
8:                 j ← j + 1
9:             i ← i + 1
10:    enlacesSwitch ← switchInicioCaminho.getEnlaces()
11:    while i < enlacesSwitch do
12:        montarCaminhosHosts(switchInicioCaminho, enlacesSwitch[i],
   switchesCaminho)
13:        i ← i + 1
```

Para contornar este problema, foi necessário adicionar ao serviço de monitoramento proposto, um mecanismo que identifica e separa os caminhos presentes entre os *hosts* ativos na rede. Dessa forma, caso ocorra alguma mudança de caminho, além de não ocorrer a perda dos dados estatísticos coletados, é possível identificar a qual caminho cada conjunto de dados pertence. A coleta destes dados, é feita através do mecanismo de *polling*, que consiste em consumir a API do controlador de tempos em tempos.

O serviço de monitoramento proposto executa uma série de etapas durante o seu funcio-

Algoritmo 3 Definição dos Fluxos

```
1: procedure DEFINIRFLUXOS()
2:     caminhosRede ← obterCaminhosRede()
3:     while i < sizeof(caminhosRede) do
4:         cadastrarFluxo(caminhosRede[i].getHostOrigem(),
   caminhosRede[i].getHostDestino(), caminhosRede[i])
5:         i ← i + 1
```

namento, com o objetivo de capturar todo o tráfego de dados da rede, e representá-lo de acordo com o modelo proposto. Estas etapas são descritas a seguir:

- Manutenção da Topologia - etapa onde são obtidas todas as informações sobre todos os *switches* presentes na rede, bem como os seus enlaces, com o objetivo de montar toda a topologia atual da rede. Além disso, são obtidos os *hosts* ativos presentes na rede, bem como a informação sobre a qual *switch* cada um se encontra conectado. O Algoritmo 1 descreve os passos realizados durante esta etapa;

- Definição dos Caminhos - uma vez identificados os *hosts* ativos da rede, é feito o mapeamento sobre os caminhos possíveis entre eles, de acordo com a topologia atual da rede. Os passos realizados durante esta etapa podem ser vistos no Algoritmo 2;

- Definição dos Fluxos - cada *host* apresenta o seu endereço IP, sendo possível identificar quais fluxos poderão atuar na rede. Com isso, são salvos todos os fluxos e os seus respectivos caminhos, com o objetivo de terem os seus dados estatísticos atualizados. O Algoritmo 3 descreve os passos realizados durante esta etapa;

- Definição dos Serviços - neste ponto, é utilizada como parâmetro uma lista contendo os serviços que serão permitidos na rede. Pelo fato de todos os *hosts* ativos já terem sido identificados, é possível salvar todos os serviços e os seus respectivos caminhos, com o objetivo de terem os seus dados estatísticos atualizados. Os passos realizados durante esta etapa podem ser vistos no Algoritmo 4;

- Coleta de Dados - é utilizado o mecanismo de *polling*, obtendo assim os dados estatísticos de cada regra presente em cada *switch*, sendo atualizados o total de pacotes e o total de *bytes* de cada fluxo e de cada serviço. O Algoritmo 5 descreve os passos realizados durante esta etapa.

5.2.2 Serviço de Balanceamento de Carga

Adicionalmente, foi desenvolvido um serviço de balanceamento de carga, que tem como principal objetivo, utilizar os dados coletados pelo serviço de monitoramento aqui proposto,

Algoritmo 4 Definição dos Serviços

```
1: procedure definirServicos(listaServicos)
2:     caminhosRede ← obterCaminhosRede()
3:     while i < sizeof(listaServicos) do
4:         while j < sizeof(caminhosRede) do
5:             cadastrarServico(caminhosRede[j].getHostOrigem(),
   caminhosRede[j].getHostDestino(), listaServicos[i].getProtocoloTransp(),
   listaServicos[i].getPorta(), caminhosRede[j])
6:             j ← j + 1
7:         i ← i + 1
```

Algoritmo 5 Coleta de Dados

```
1: procedure coletarDados()
2:     caminhosRede ← obterCaminhosRede()
3:     while i < sizeof(caminhosRede) do
4:         cadastrarFluxo(caminhosRede[i].getHostOrigem(),
   caminhosRede[i].getHostDestino(), caminhosRede[i])
5:         i ← i + 1
```

e alterar o estado da rede, visando uma melhora na sua utilização. Este serviço foi desenvolvido através da linguagem de programação Java, e atua como um subserviço do serviço de monitoramento de tráfego.

Para aplicar o balanceamento de carga, foi necessário não somente conhecer todos os caminhos da rede, mas também ter a capacidade de alterar os fluxos e os serviços presentes em cada um deles, caso isso fosse necessário. Para tal, foi utilizado o seguinte objeto remoto provido pela API do OpenDaylight:

- FlowProgrammerNorthbound - através deste objeto, foi possível obter informações sobre todas as regras cadastradas em cada um dos *switches* da rede, permitindo alterá-las, removê-las, ou adicionar novas, caso seja necessário, tudo com o objetivo de alterar o caminho de um determinado fluxo ou serviço.

O serviço de balanceamento de carga, apresenta três abordagens distintas, com relação à escolha de qual caminho deverá ser utilizado por cada um dos fluxos e serviços presentes na rede. Para tomar esta decisão, foram implementados os seguintes algoritmos de escalonamento:

1. Round-Robin - algoritmo implementado através de uma fila circular, tradicionalmente utilizado pelos sistemas operacionais para realizar o escalonamento de processos. Neste contexto, cada processo ocupa uma posição da fila, sendo executado pelo processador durante um determinado período de tempo. Após o término deste intervalo de tempo, é executado o próximo processo da fila. Este ciclo se mantém até que não exista mais nenhum processo para ser executado. Trazendo este conceito para o contexto aqui analisado, a fila

Algoritmo 6 Balanceamento de Carga

```
1:  procedure aplicarBalanceamentoCarga(tipoBalanceamento)
2:      switch tipoBalanceamento do
3:          case roundRobin
4:              aplicarRoundRobin()
5:          case bandwidthBased
6:              aplicarBandwidthBased()
7:          case roundRobinAndBandwidthBased
8:              aplicarRoundRobinBandwidthBased()
9:          case nenhum
10:             exit()
```

cíclica é composta pelos caminhos da rede, sendo assim, sempre que um novo fluxo ou serviço precise ser alocado em um determinado caminho, é escolhido o primeiro da fila, garantindo uma distribuição uniforme entre todos os caminhos da rede;

2. *Bandwidth-Based* - assim como o Round-Robin, este algoritmo foi utilizado para determinar em qual caminho um determinado fluxo/serviço deverá seguir. A principal diferença com relação ao Round-Robin, é o fato de não alocar um fluxo/serviço no primeiro caminho disponível. Para isso, é feita uma relação entre a utilização atual de cada caminho (total de *bytes* trafegados em um determinado instante) e um limiar máximo de utilização definido (número que indica o percentual máximo de utilização que cada caminho pode atingir). Por exemplo, foi definido como limiar de utilização dos caminhos, um valor máximo de 70%, e existem dois caminhos na rede, um deles está com 40% de utilização, e o outro com 60%. Caso um novo fluxo/serviço precise ser alocado em algum caminho, o caminho com 40% de utilização será escolhido;

3. Round-Robin + *Bandwidth-Based* - apesar da política de atendimento parecer justa, o Round-Robin pode acabar alocando fluxos/serviços de maior carga em um mesmo caminho, visto que ele não leva em consideração o tráfego gerado na rede para fazer as alocações, tornando-se ineficiente. Em contrapartida, calcular a utilização dos caminhos para aplicar a alocação, pode ser algo custoso. Com isso, foi proposta uma união entre os dois algoritmos, que visa utilizar o melhor de cada um deles, fazendo uma alocação inicial utilizando o Round-Robin, e aplicando posteriormente o *Bandwidth-Based* caso seja necessário.

O Algoritmo 6 descreve as etapas realizadas durante a execução do serviço de balanceamento de carga. Para executar o serviço, é necessário informar qual tipo de balanceamento será aplicado à rede.

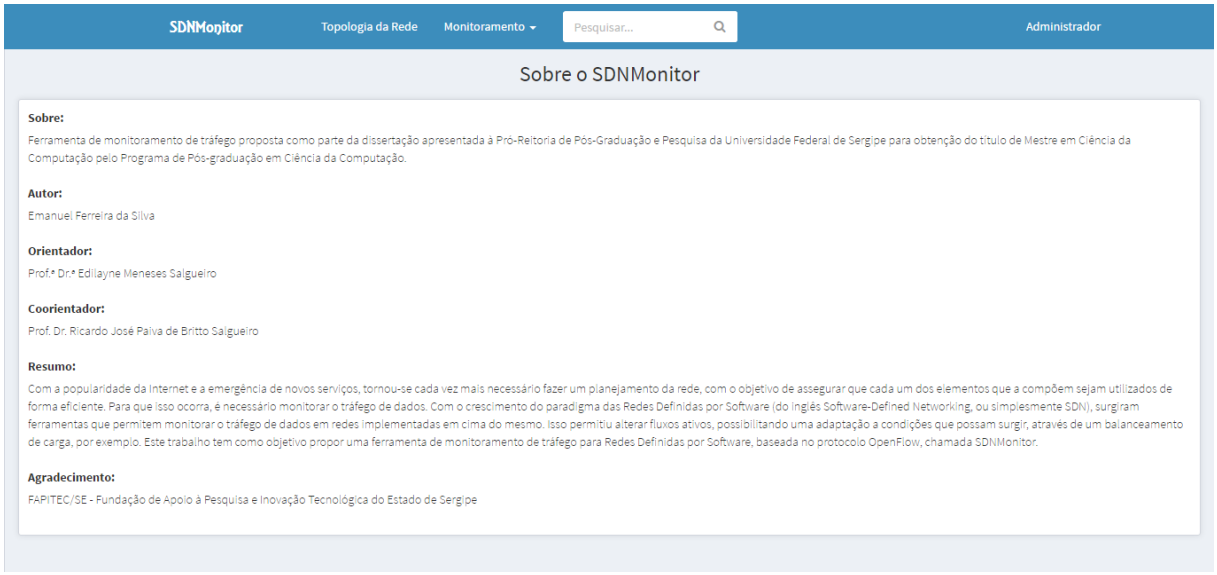


Figura 5.3 – Página inicial da aplicação Web. Fonte: Criada pelo autor

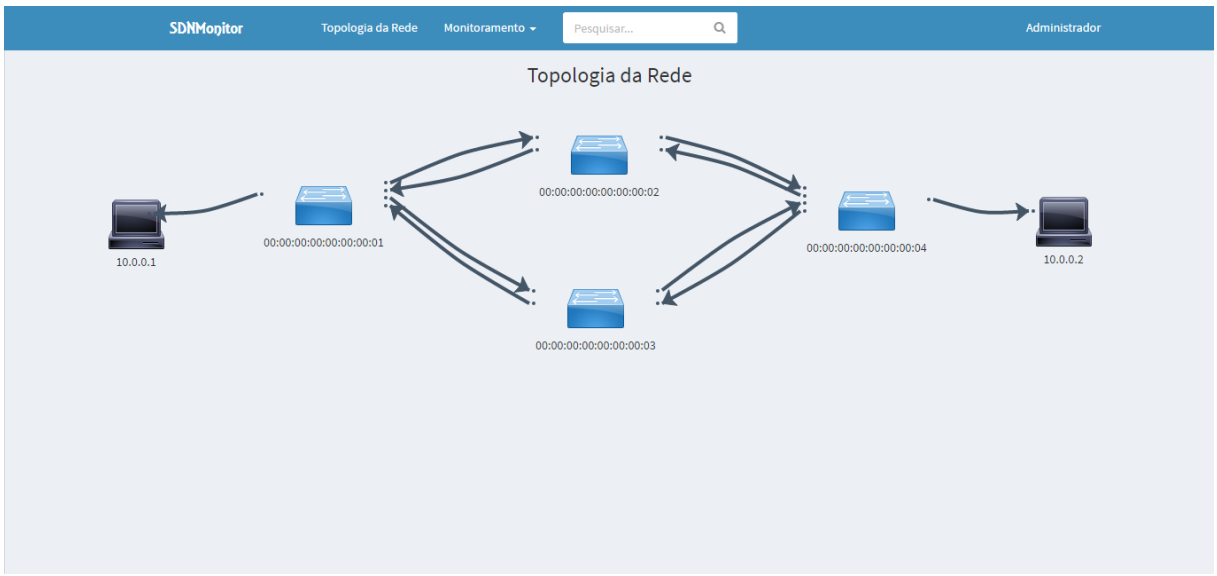


Figura 5.4 – Página da topologia da rede. Fonte: Criada pelo autor

5.3 Camada de Aplicações

Com o objetivo de facilitar o processo de gerenciamento da rede, foi desenvolvida uma aplicação Web, que tem como principal objetivo fornecer uma visão da rede em vários níveis de granularidade, tais como: (i) tráfego de dados em cada porta de cada um dos *switches* da rede, (ii) tráfego de dados em cada fluxo da rede e (iii) o tráfego de dados a nível de serviço.

Para desenvolver a aplicação, foi utilizada a linguagem de programação PHP para o *back-end*, e a linguagem de programação JavaScript para o *front-end*, juntamente com HTML e CSS. A Figura 5.3 apresenta a tela inicial da aplicação, na qual são apresentadas algumas informações básicas sobre o mesmo, tais como o objetivo da aplicação, o autor, o orientador, o coorientador e o resumo deste trabalho, além de um agradecimento a FAPITEC/SE (Fundação de

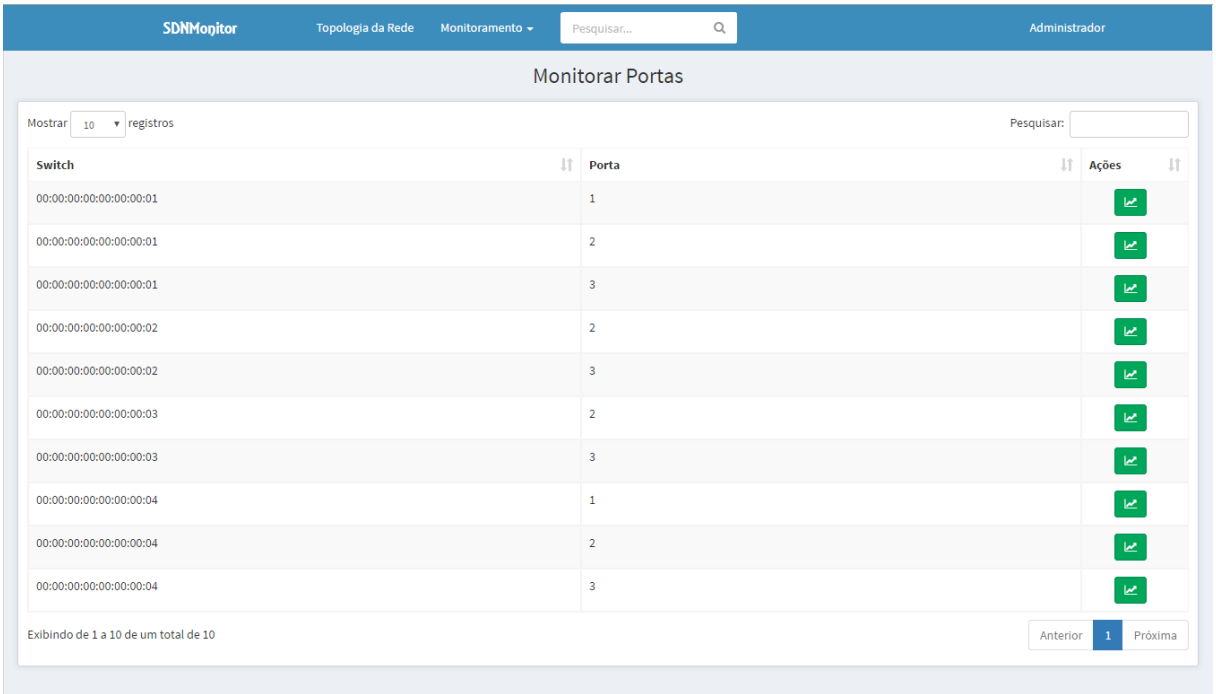


Figura 5.5 – Página dos *switches* e respectivas portas para monitoramento. Fonte: Criada pelo autor

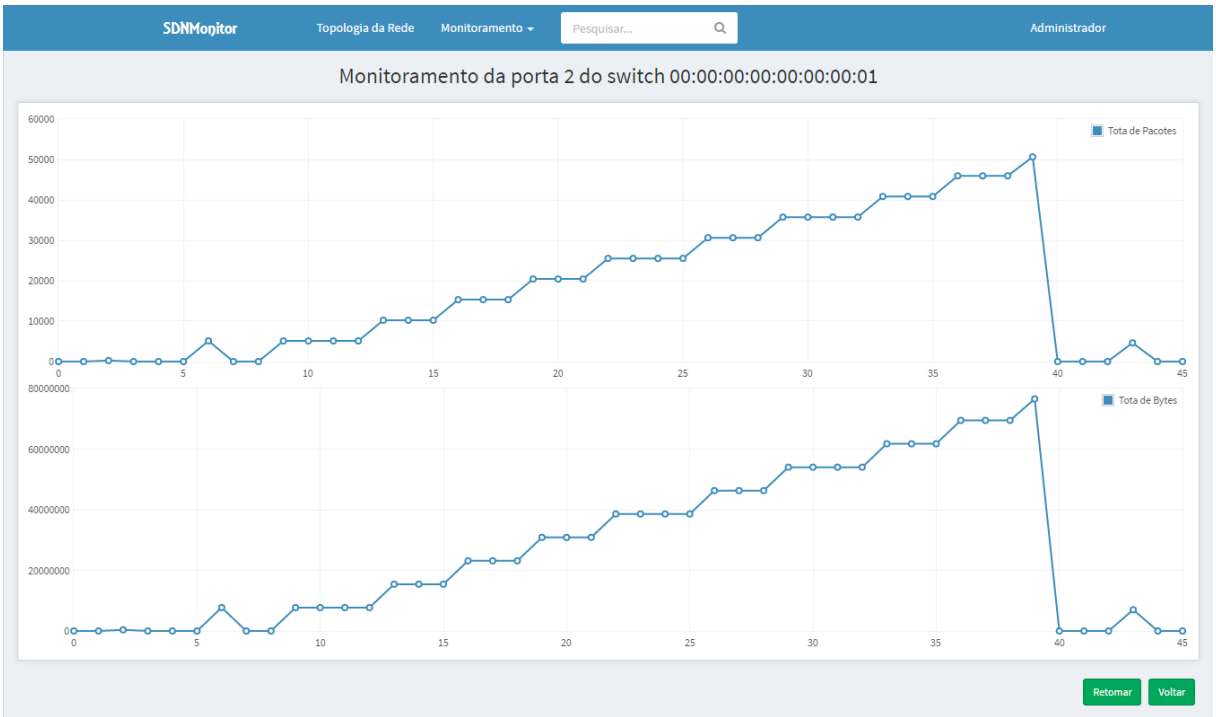


Figura 5.6 – Página de monitoramento a nível de porta do *switch*. Fonte: Criada pelo autor

Apoio à Pesquisa e Inovação Tecnológica do Estado de Sergipe) que financiou parte do projeto. As demais telas da aplicação são utilizadas para obter informações sobre a rede, e consistem nas seguintes:

- Topologia da Rede - funcionalidade responsável por apresentar a topologia da rede de

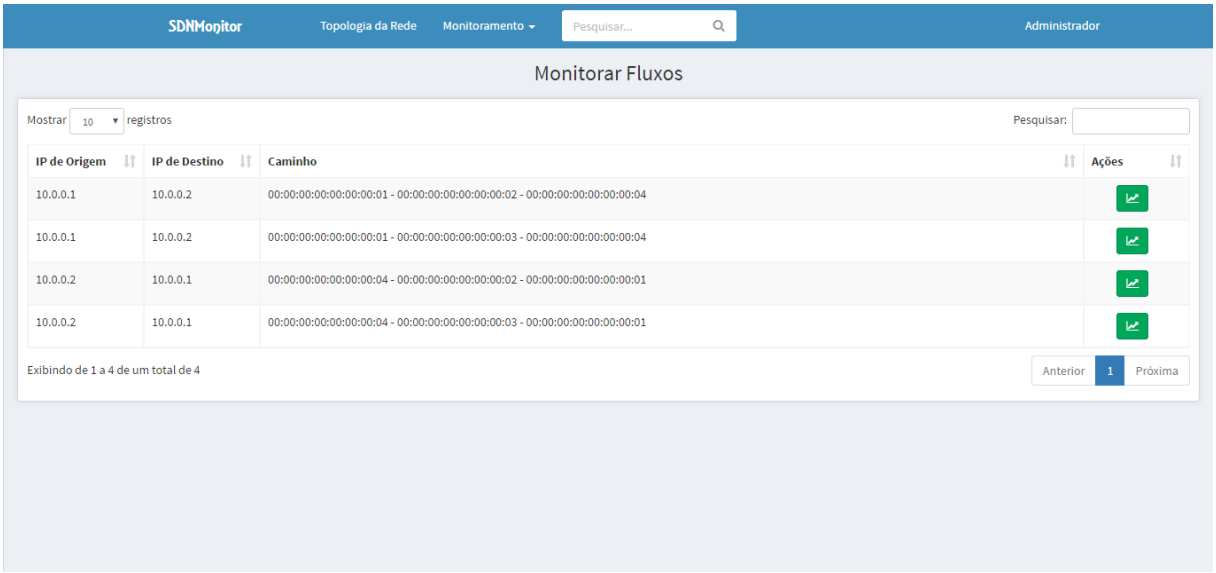


Figura 5.7 – Página dos fluxos para monitoramento. Fonte: Criada pelo autor

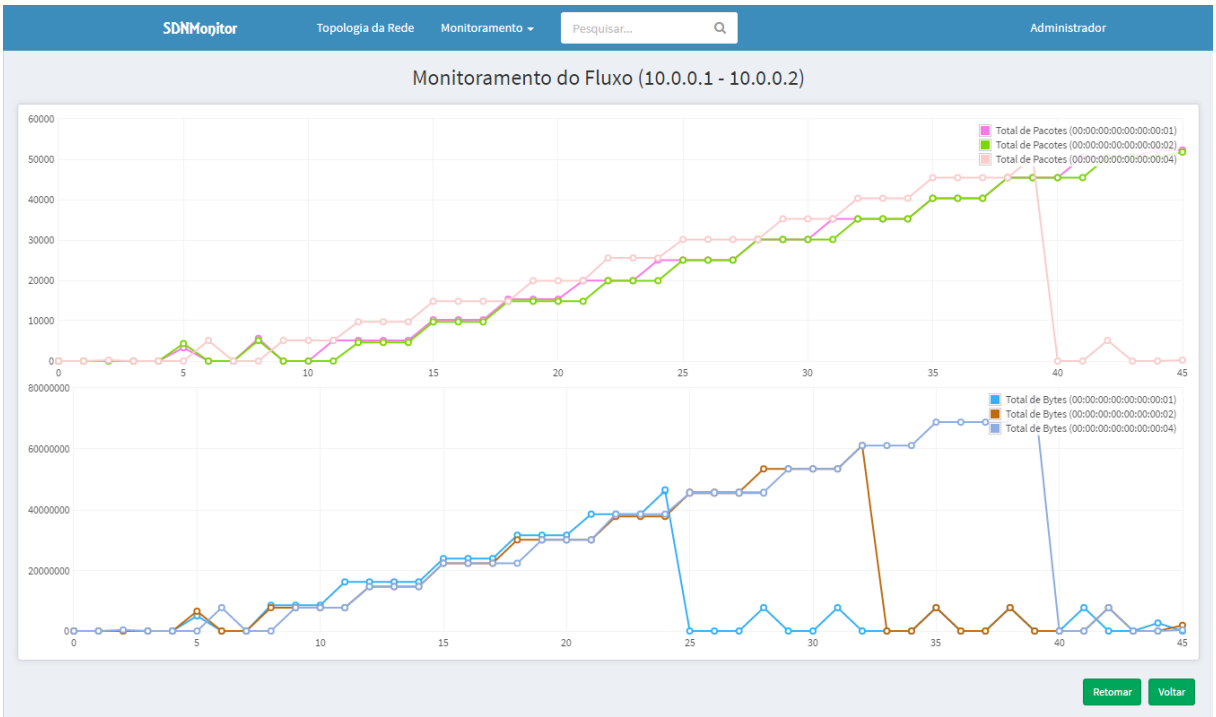


Figura 5.8 – Página de monitoramento a nível de fluxo. Fonte: Criada pelo autor

forma gráfica, facilitando a identificação dos nós descobertos na rede (*switches*, *hosts* e etc.). A Figura 5.4 apresenta uma exemplo de como seria exibida uma topologia com 4 *switches* e 2 *hosts*;

- Monitoramento por Porta - funcionalidade responsável por obter todos os *switches* da rede e suas respectivas portas, com o objetivo de permitir ao administrador monitorar o tráfego de dados (pacotes e *bytes*) em cada uma delas. A Figura 5.5 apresenta um exemplo disso, onde é possível ver 4 *switches* e suas respectivas portas. O monitoramento do tráfego em uma determinada porta (nesse caso a de número 2) de um determinado *switch* (nesse caso

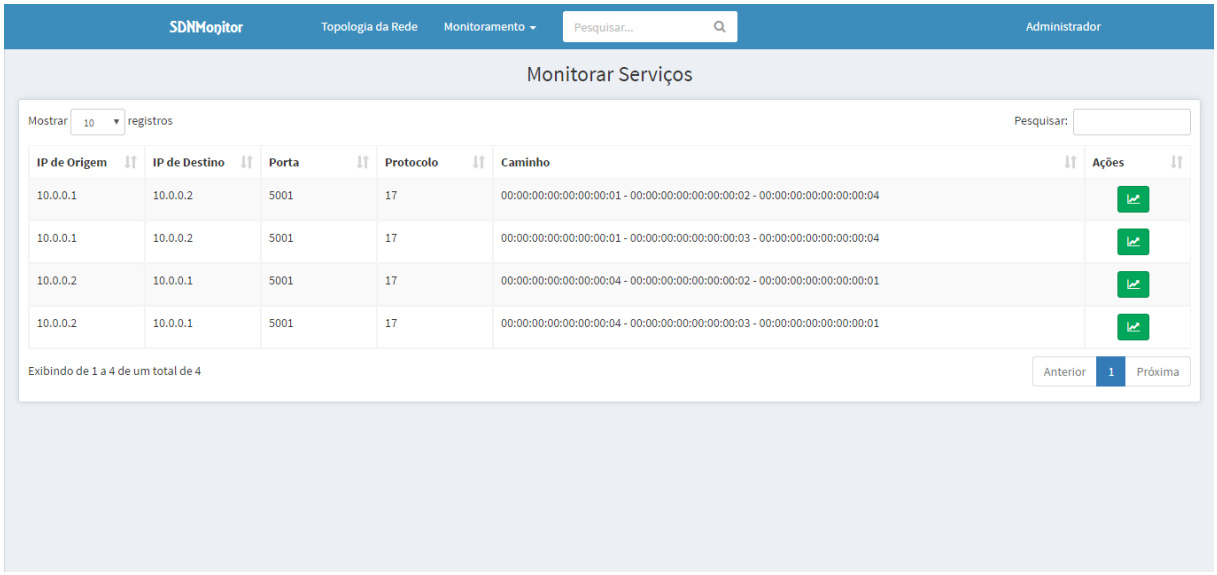


Figura 5.9 – Página dos serviços para monitoramento. Fonte: Criada pelo autor

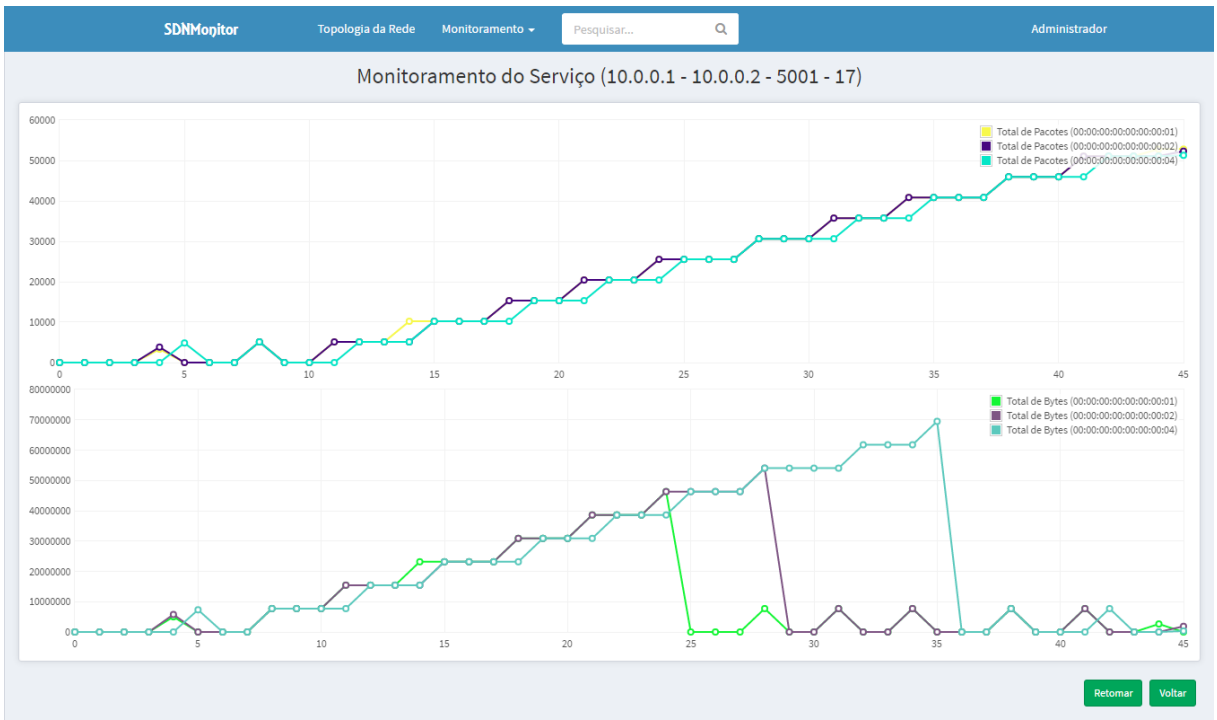


Figura 5.10 – Página de monitoramento a nível de serviço. Fonte: Criada pelo autor

o 00:00:00:00:00:00:00:01) pode ser visto na Figura 5.6;

- Monitoramento por Fluxo - funcionalidade responsável por obter todos os fluxos da rede, com o objetivo de permitir ao administrador monitorar o tráfego de dados (pacotes e *bytes*) em cada um deles. A Figura 5.7 apresenta 4 exemplos de fluxos, onde é possível perceber em ambos a presença do IP de origem, IP de destino e o caminho por onde passa cada um deles. O monitoramento do tráfego de um determinado fluxo (10.0.0.1 - 10.0.0.2) pode ser visto na Figura 5.8;

- Monitoramento por Serviço - funcionalidade responsável por obter todos os serviços da rede, com o objetivo de permitir ao administrador monitorar o tráfego de dados (pacotes e *bytes*) em cada um deles. A Figura 5.9 apresenta 2 exemplos de serviços, onde é possível perceber em ambos a presença do IP de origem, IP de destino, porta utilizada, protocolo da camada de transporte (nesse caso o UDP, representado pelo número 17 como explicado na Seção 5.2) e o caminho por onde passa cada um deles. O monitoramento do tráfego de um determinado serviço (10.0.0.1 - 10.0.0.2 - 5001 - 17) pode ser visto na Figura 5.10;

Todos os dados apresentados pela aplicação Web, são obtidos através de consultas realizadas em uma base de dados, que foi criada de acordo com o modelo definido na Seção 5.1. Estes dados são inseridos na base pelo serviço de monitoramento, descrito na Seção 5.2.

6 Casos de Uso

A ocorrência de um grande número de fluxos simultâneos em um caminho degradam o desempenho da rede. A adoção de mecanismos de monitoramento de tráfego, aliado a mecanismos de balanceamento de carga tem impacto na vazão e utilização da rede.

Neste capítulo são apresentados experimentos de monitoramento de redes para validar os serviços desenvolvidos nesta dissertação.

6.1 Cenário Proposto

O sistema avaliado nesse capítulo é composto por uma rede SDN com um controlador Opendaylight monitorando *switches* OpenFlow. Foram realizados experimentos de medição para avaliar o mecanismo de *polling* do monitoramento de tráfego e o mecanismo de mudança de caminhos do serviço de balanceamento de carga.

A infraestrutura de rede foi instanciada no Mininet versão 2.2.0, executando o Open vSwitch versão 2.0.2 nos elementos de rede. Foi utilizado o controlador OpenDaylight versão Hydrogen, juntamente com os serviços de monitoramento e balanceamento de carga aqui propostos. Todos os enlaces da rede foram criados com uma largura de banda igual a 10 Mbps. Os experimentos foram efetuados em um Intel Core i5-4210U CPU @ 1.70GHz 1.70GHz com 8 GB de memória RAM e sistema operacional Windows 10. Vale a pena destacar que tanto o controlador quanto os serviços foram executados na máquina física. E o Mininet em uma máquina virtual com um processador e 2 GB de memória RAM, com o sistema operacional Ubuntu 14.04 LTS.

Foram propostas duas topologias distintas para a realização dos experimentos. A primeira delas (ver a Figura 6.1) foi composta por quatro *switches* (SW1, SW2, SW3 e SW4) e doze *hosts* (H1, H2, H3, H4, H5, H6, H7, H8, H9, H10, H11 e H12), com dois possíveis caminhos entre eles. Já a segunda (ver a Figura 6.2), foi composta por oito *switches* (SW1, SW2, SW3, SW4, SW5, SW6, SW7 e SW8), com os mesmos doze *hosts* da primeira, mas com três opções de caminho.

Com o objetivo de avaliar o desempenho da rede durante a execução dos experimentos, foram levados em consideração a taxa de transmissão de cada um dos tráfegos gerados e de cada um dos caminhos da rede, e a variação do atraso (ou seja, o Jitter) para os tráfegos que utilizam o protocolo UDP na camada de transporte.

Para representar a geração de carga, foram gerados três tipos de tráfego, com objetivo de representar três tipos de aplicações distintas. Tipicamente, aplicações de vídeo utilizam o protocolo UDP na camada de transporte, o que ocorre devido ao fato do mesmo ser mais rápido do que o TCP, visto que não faz nenhum tipo de controle de tráfego, e tem como uma de suas

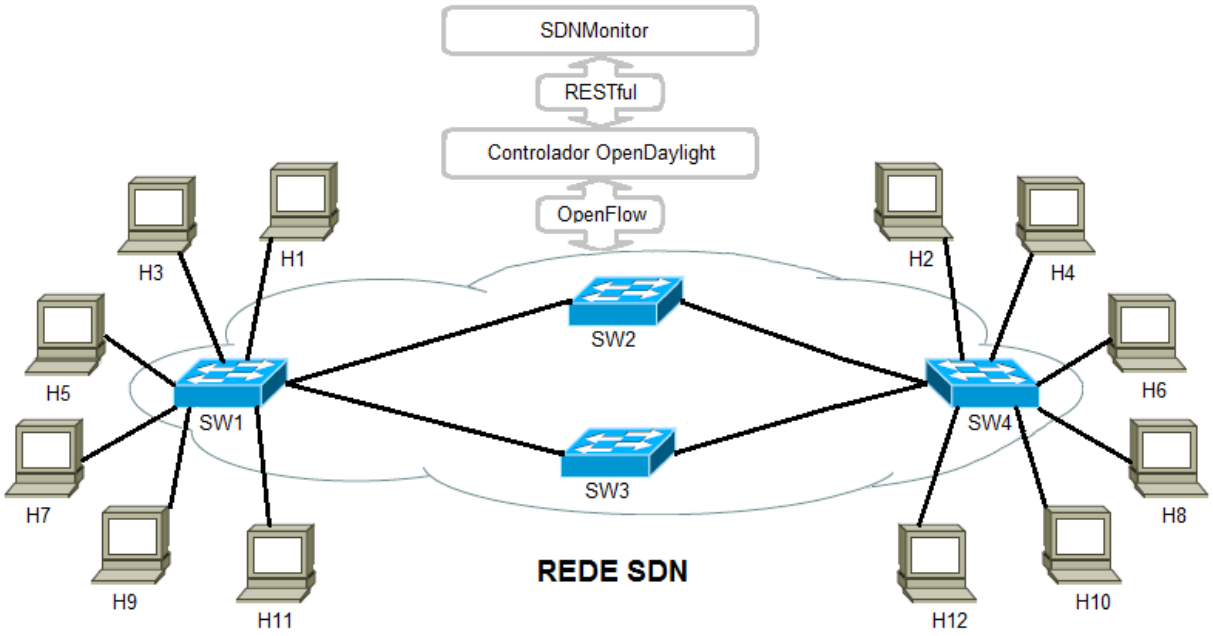


Figura 6.1 – Topologia de rede utilizada para a realização dos experimentos com quatro *switches* e dois caminhos. Fonte: Criada pelo autor.

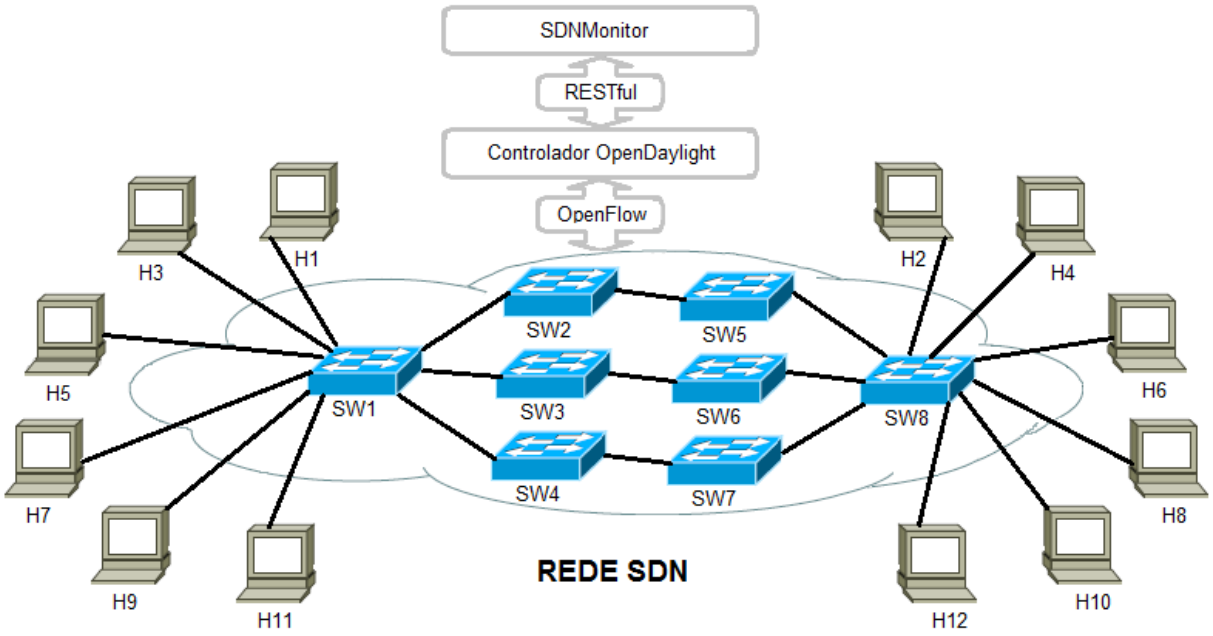


Figura 6.2 – Topologia de rede utilizada para a realização dos experimentos com oito *switches* e três caminhos. Fonte: Criada pelo autor.

características tentar consumir toda a banda de rede disponível (ZHENG; BOYCE, 2001). Foi gerado então um tráfego UDP representando uma aplicação de vídeo (chamado a partir de agora de tVideo).

Apesar de realizar um certo controle, existem alguns tráfegos TCP que podem apresentar um comportamento parecido com alguns tráfegos UDP, como é o caso dos elefantes, que apesar de serem TCP, podem apresentar picos de transmissão (BROWNLEE; CLAFFY, 2002). O

segundo tráfego então, consiste em um tráfego TCP tipicamente classificado como elefante (chamado a partir de agora de tElefante). Por fim, o terceiro tráfego gerado também consiste em um tráfego TCP, porém com uma carga menor, tipicamente apresentado por aplicações Web (MOLINA; CASTELLI; FODDIS, 2000). Todos os tipos de tráfegos foram gerados através do Iperf (TIRUMALA et al., 2005).

6.2 Experimentos

Os experimentos foram divididos em duas partes, na primeira delas é proposta uma comparação entre uma abordagem sem balanceamento de carga (Cenário 1) com as técnicas de balanceamento de carga Round-Robin (Cenário 2) e *Bandwidth-Based* (Cenário 3). Já na segunda parte, é proposta uma comparação com um cenário sem nenhum tipo de balanceamento de carga (Cenário 4) com um cenário onde as técnicas de balanceamento Round-Robin e *Bandwidth-Based* foram aplicadas juntas (Cenário 5). Os cenários de teste são descritos a seguir:

- Cenário 1 (sem balanceamento de carga) - cenário onde foi gerado um tráfego do tipo tVideo entre os *hosts* H1 e H2 durante 120 segundos, sendo que após os primeiros 30 segundos foi gerado um segundo tráfego do tipo tElefante entre os *hosts* H3 e H4 durante 90 segundos, com o objetivo de gerar uma concorrência pelos recursos da rede. Durante os 120 segundos nenhuma técnica de balanceamento de carga foi aplicada;
- Cenário 2 (Round-Robin) - assim como no Cenário 1 foram gerados dois tráfegos nas mesmas condições entre os *hosts* H1, H2, H3 e H4, com a diferença de que foi utilizado o algoritmo Round-Robin no serviço de balanceamento de carga;
- Cenário 3 (*Bandwidth-Based*) - cenário parecido com o Cenário 2, com a diferença de que o algoritmo de balanceamento de carga utilizado foi o *Bandwidth-Based*, com um limiar máximo de utilização de 90%;
- Cenário 4 (sem balanceamento de carga) - cenário onde foi gerado um tráfego do tipo tVideo entre os *hosts* H1 e H2 durante 120 segundos, sendo que após os primeiros 30 segundos foi gerado um segundo tráfego do tipo tElefante entre os *hosts* H3 e H4 durante 90 segundos, assim como nos cenários anteriores, com a diferença de que após os primeiros 60 segundos foram gerados quatro tráfegos adicionais do tipo tWeb entre os *hosts* H5, H6, H7, H8, H9, H10, H11 e H12. Durante os 120 segundos nenhuma técnica de balanceamento de carga foi aplicada;
- Cenário 5 (Round-Robin + *Bandwidth-Based*) - assim como no Cenário 4 foram gerados dois tráfegos nas mesmas condições entre todos os *hosts* ativos da rede, com a diferença de que foram utilizados os algoritmos de balanceamento de carga Round-Robin e *Bandwidth-Based* (limiar de utilização de 90%, como no Cenário 3) no serviço de balanceamento de carga.

Para analisar os cenários propostos, foram coletadas as taxas de transmissão de todos os tráfegos gerados, além da utilização de cada um dos caminhos em ambas as topologias de rede experimentadas, que consiste no total de *bytes* trafegando por cada um deles. Os caminhos presentes na rede consistem nos seguintes:

- Caminho 1 - presente na primeira topologia e composto pelos *switches* SW1, SW2 e SW4;
- Caminho 2 - presente na primeira topologia e composto pelos *switches* SW1, SW3 e SW4;
- Caminho 3 - presente na segunda topologia e composto pelos *switches* SW1, SW2, SW5 e SW8;
- Caminho 4 - presente na segunda topologia e composto pelos *switches* SW1, SW3, SW6 e SW8;
- Caminho 5 - presente na segunda topologia e composto pelos *switches* SW1, SW4, SW7 e SW8.

Adicionalmente, foram configurados agentes sFlow nos *switches* da rede, assim como um coletor, com o objetivo de capturar amostras do tráfego da rede, e como o protocolo sFlow se comporta atuando junto à solução de monitoramento proposta.

6.3 Resultados

Os resultados aqui apresentados são frutos de um experimento realizado em duas partes. Em ambas, foram coletados a vazão dos fluxos e serviços presentes na rede, e a utilização de cada um dos caminhos da rede. Foram monitorados todos os *switches* presentes nas topologias utilizadas para a realização dos experimentos.

As Figuras 6.3 e 6.4, apresentam a taxa de transmissão coletada em função do tempo no Cenário 1, onde é possível perceber que o tráfego tVideo tenta consumir toda a banda de rede disponível, interferindo diretamente no desempenho do tráfego tElefante, o que ocorreu pelo fato de não haver nenhum mecanismo de balanceamento de carga atuando na rede. Já no Cenário 2, onde foi aplicado o algoritmo de balanceamento Round-Robin, foi possível perceber que apesar de consumir toda a banda disponível, o tráfego tVideo não interferiu no tElefante (ver as Figuras 6.5 e 6.6). É possível perceber um pequeno pico na Figura 6.6 entre os instantes 25 e 27, causado pelo tempo que o algoritmo levou para identificar que algum caminho estava com um fluxo a mais.

Com relação à aplicação do algoritmo *Bandwidth-Based* (Cenário 3), foi possível perceber um comportamento bastante semelhante em ambas as topologias, como mostram as Figuras 6.7 e 6.8, onde levou-se aproximadamente cinco segundos para o algoritmo identificar que a rede estava desbalanceada, e alterar o caminho de um dos fluxos.

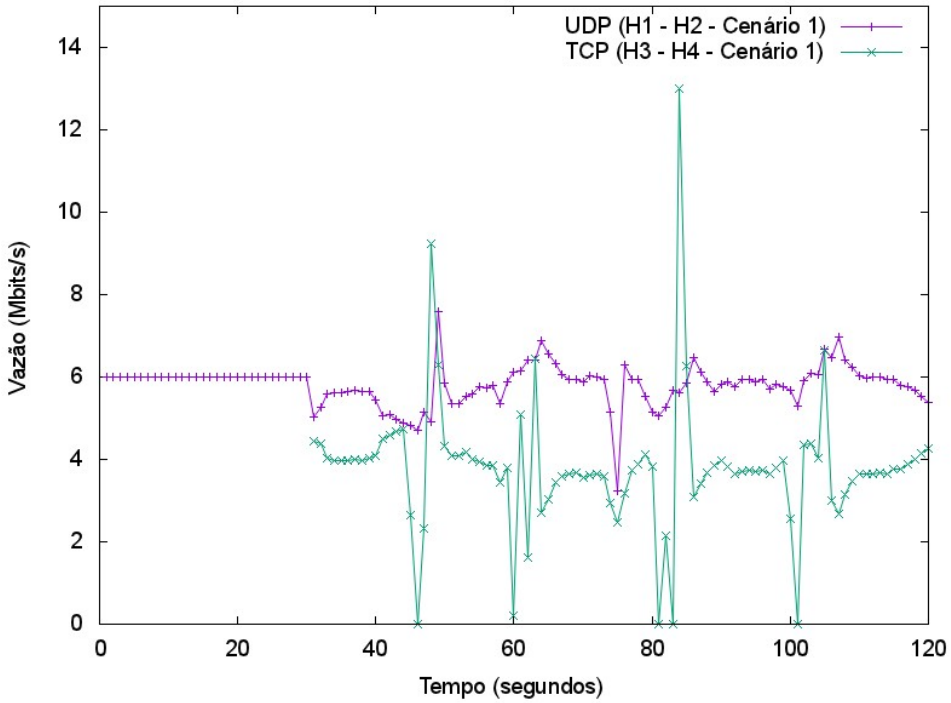


Figura 6.3 – Taxa de transmissão obtida para os tráfegos gerados no Cenário 1 com quatro *switches*. Fonte: Criada pelo autor.

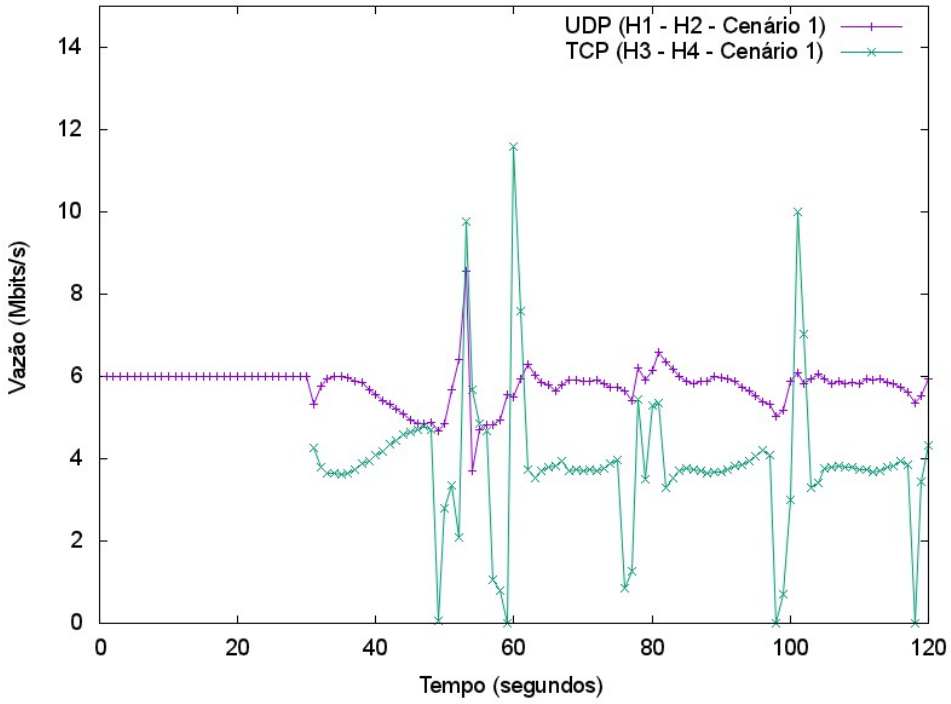


Figura 6.4 – Taxa de transmissão obtida para os tráfegos gerados no Cenário 1 com oito *switches*. Fonte: Criada pelo autor.

A Tabela 6.1 apresenta a média da taxa de transmissão e a melhora obtida em cada um dos tráfegos gerados nos cenários 1, 2 e 3, levando em consideração ambas as topologias experimentadas. É possível perceber que, nos cenários com o balanceamento de carga, houve um

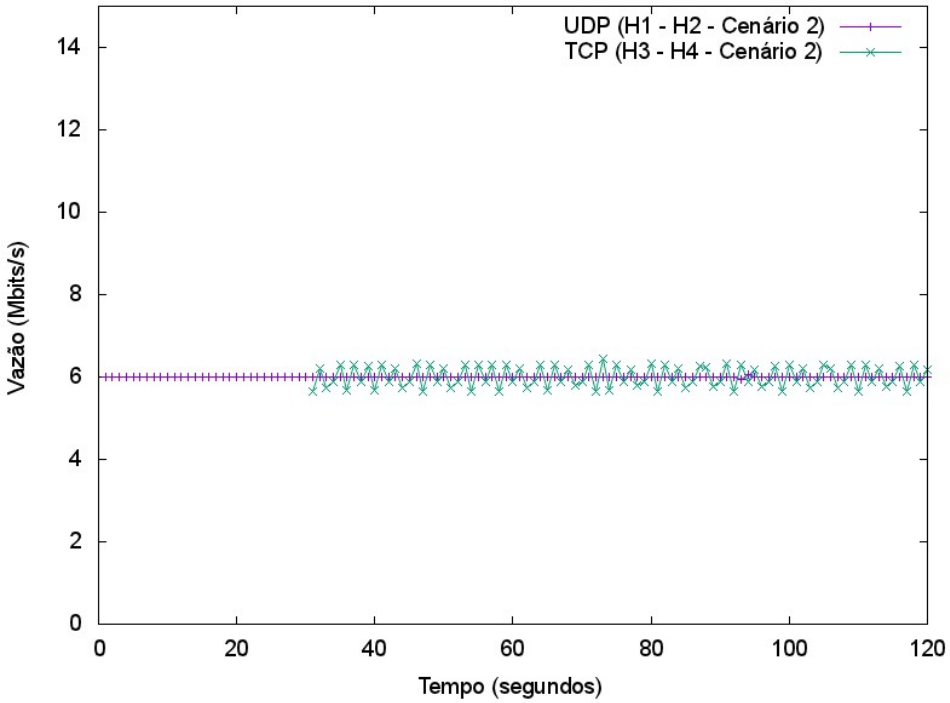


Figura 6.5 – Taxa de transmissão obtida para os tráfegos gerados no Cenário 2 com quatro *switches*. Fonte: Criada pelo autor.

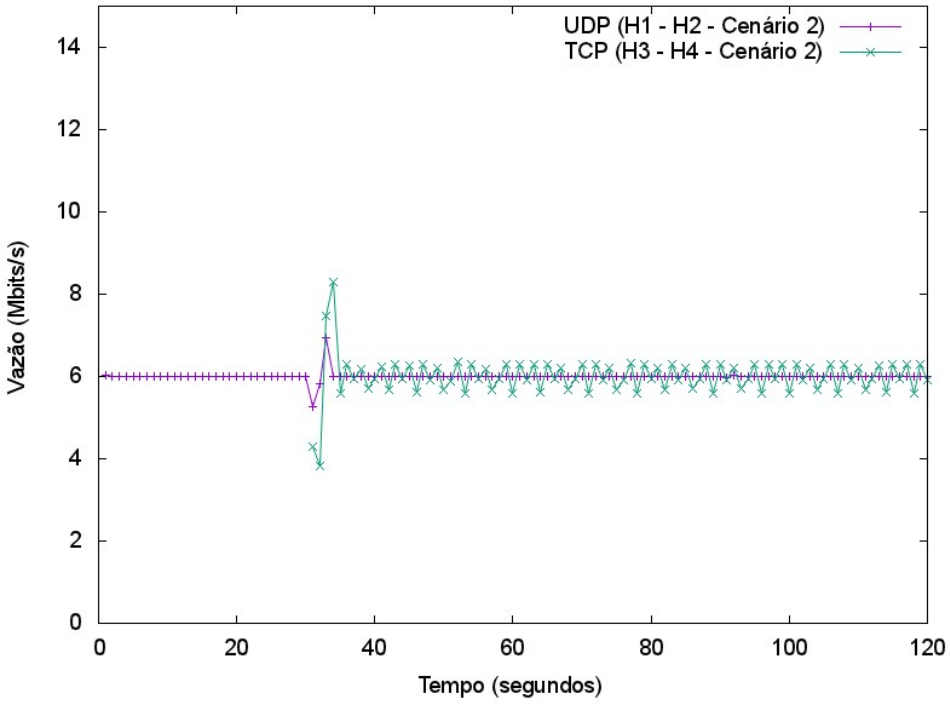


Figura 6.6 – Taxa de transmissão obtida para os tráfegos gerados no Cenário 2 com oito *switches*. Fonte: Criada pelo autor.

aumento na taxa de transmissão média de quase 60% para os tráfegos tElefante. Com relação a utilização dos caminhos da rede, tanto com 4 quanto com 8 switches, pôde-se observar que no Cenário 1 (sem balanceamento) houve uma sobrecarga com relação aos caminhos 1 e 4, e uma

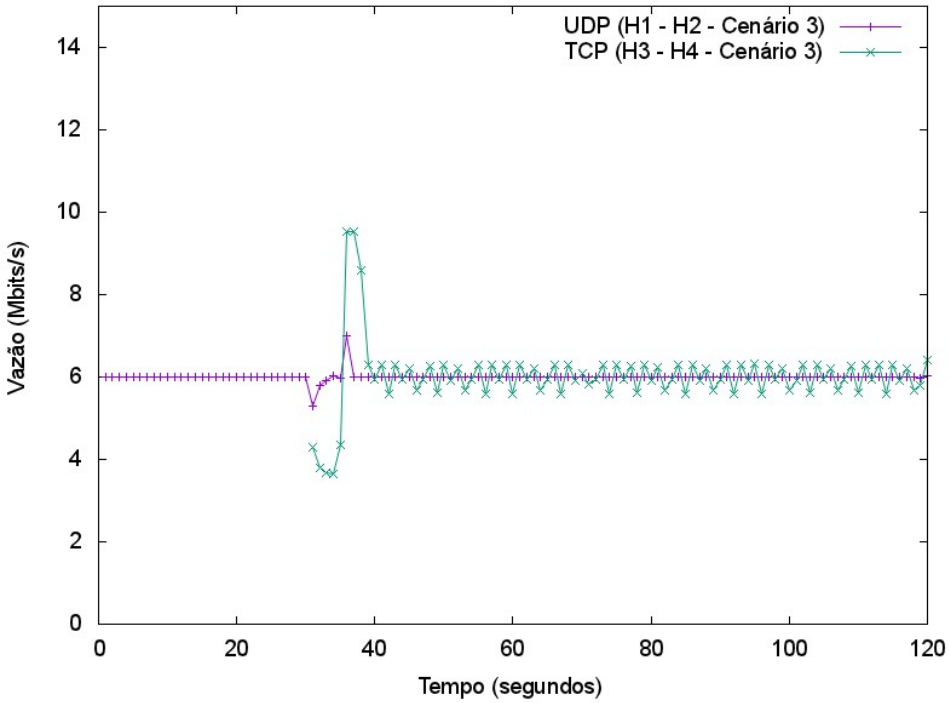


Figura 6.7 – Taxa de transmissão obtida para os tráfegos gerados no Cenário 3 com quatro *switches*. Fonte: Criada pelo autor.

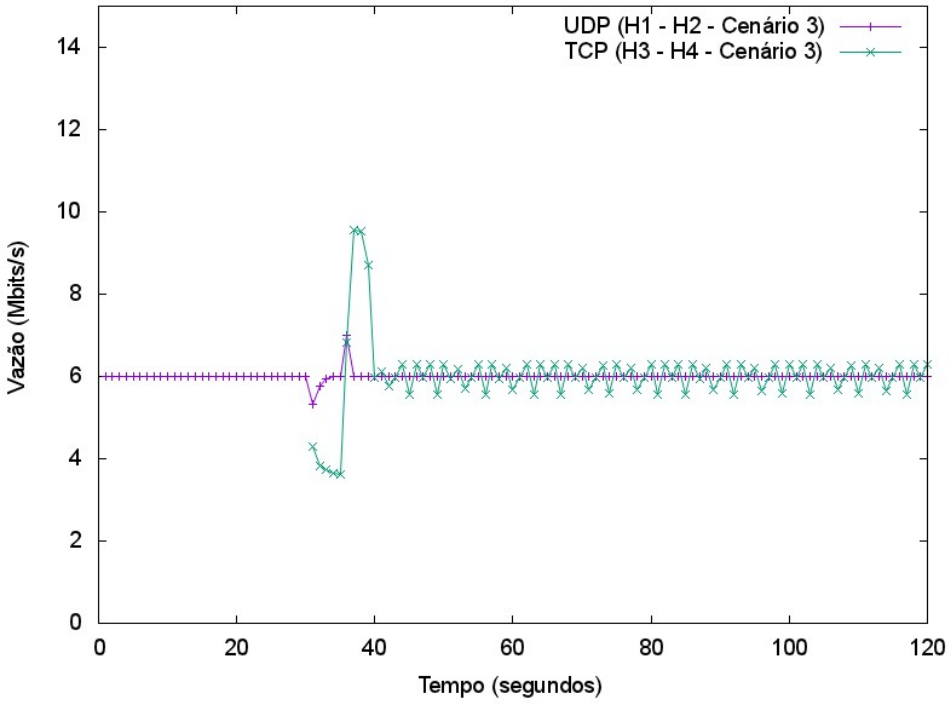


Figura 6.8 – Taxa de transmissão obtida para os tráfegos gerados no Cenário 3 com oito *switches*. Fonte: Criada pelo autor.

subutilização dos demais. Enquanto que nos cenários com balanceamento (Cenário 2 e Cenário 3), houve uma melhor distribuição do tráfego entre os caminhos (ver as Figuras 6.9 e 6.10).

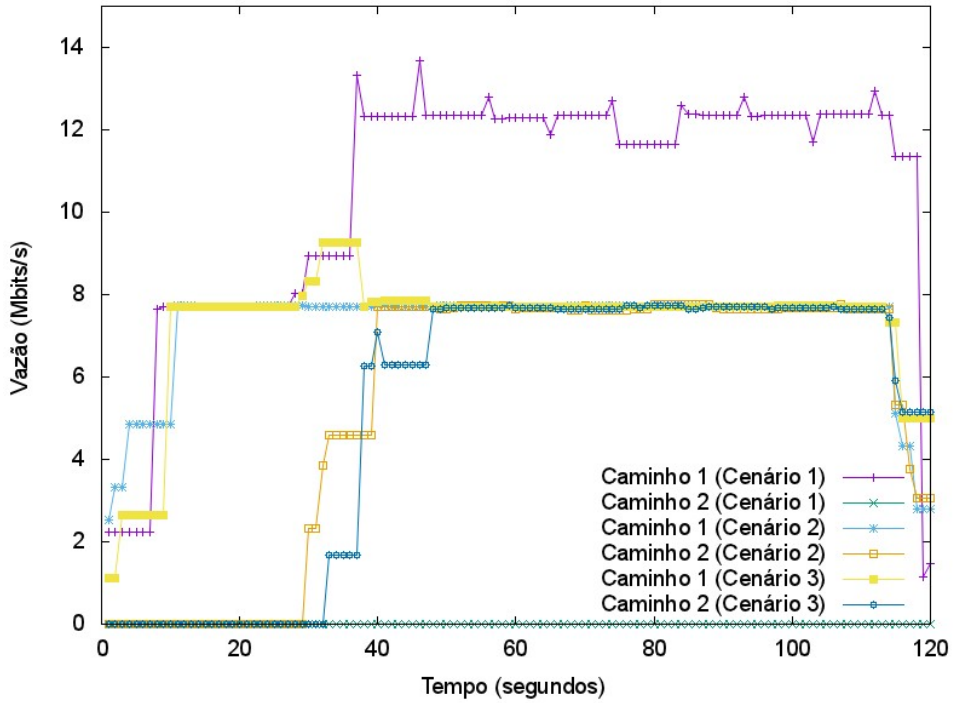


Figura 6.9 – Utilização dos caminhos da rede nos cenários 1, 2, e 3 com quatro *switches*. Fonte: Criada pelo autor.

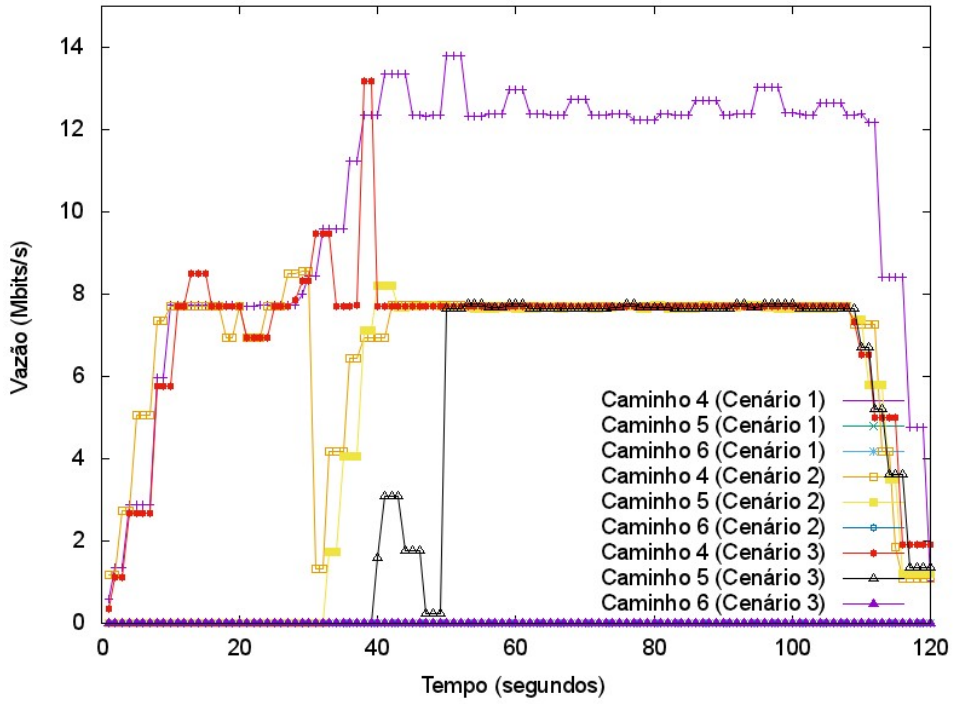


Figura 6.10 – Utilização dos caminhos da rede nos cenários 1, 2, e 3 com oito *switches*. Fonte: Criada pelo autor.

Assim como na primeira parte dos experimentos, o cenário com a aplicação das técnicas de balanceamento apresentou melhores resultados. As Figuras 6.11 e 6.12 apresentam a taxa de transmissão para os tráfegos gerados no Cenário (4), onde é possível perceber que, além da inter-

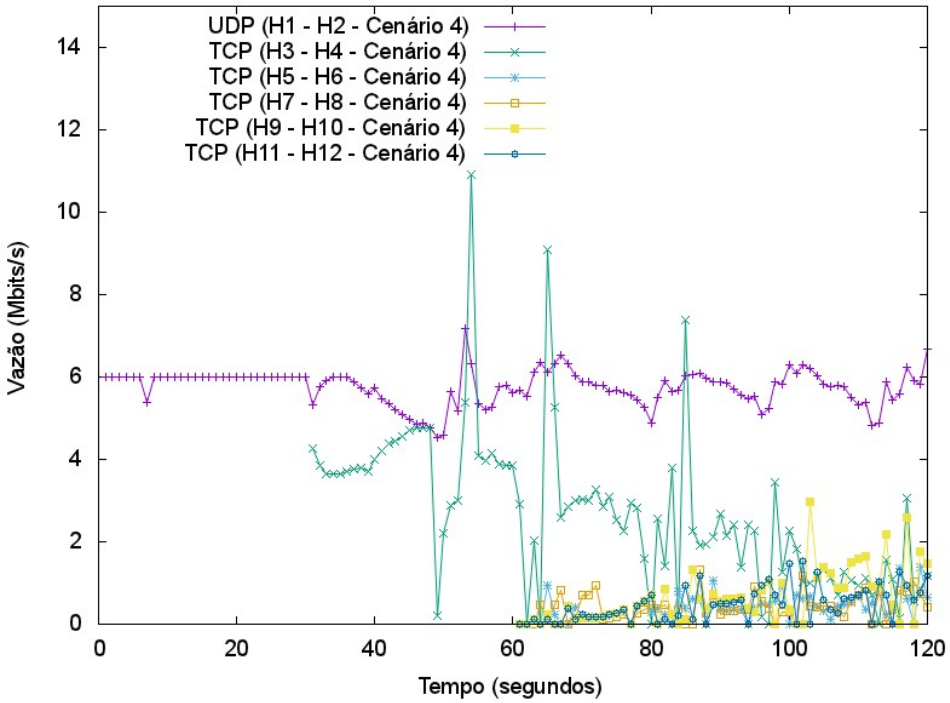


Figura 6.11 – Taxa de transmissão obtida para os tráfegos gerados no Cenário 4 com quatro *switches*. Fonte: Criada pelo autor.

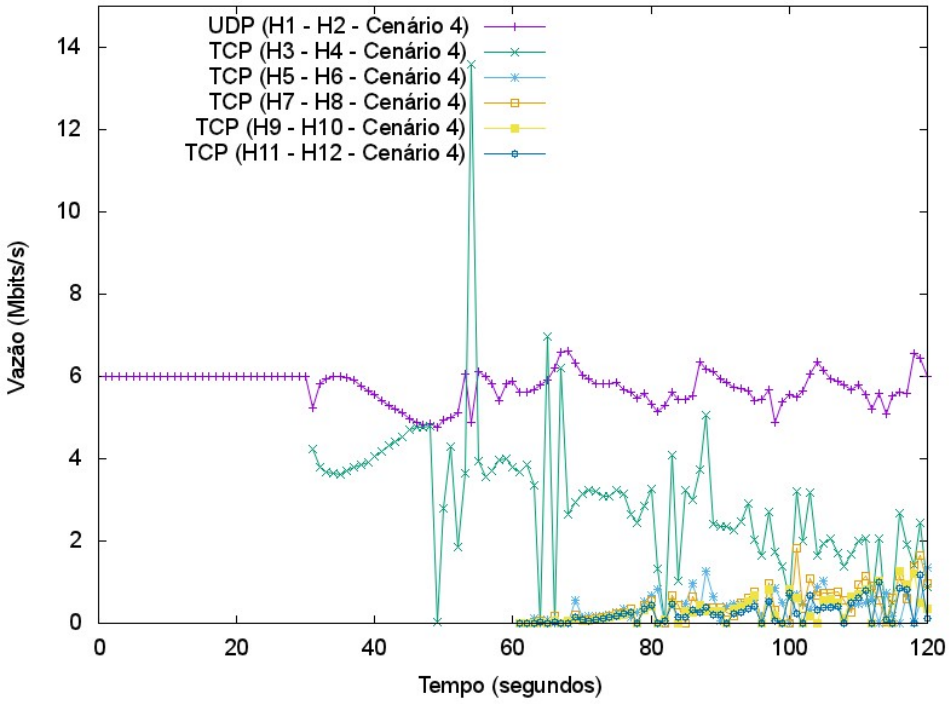


Figura 6.12 – Taxa de transmissão obtida para os tráfegos gerados no Cenário 4 com oito *switches*. Fonte: Criada pelo autor.

ferência causada pelo tVideo, o tElefante sofre uma degradação muito grande no momento em que são inseridos os tráfegos tWeb, e que estes também não apresentaram um bom desempenho.

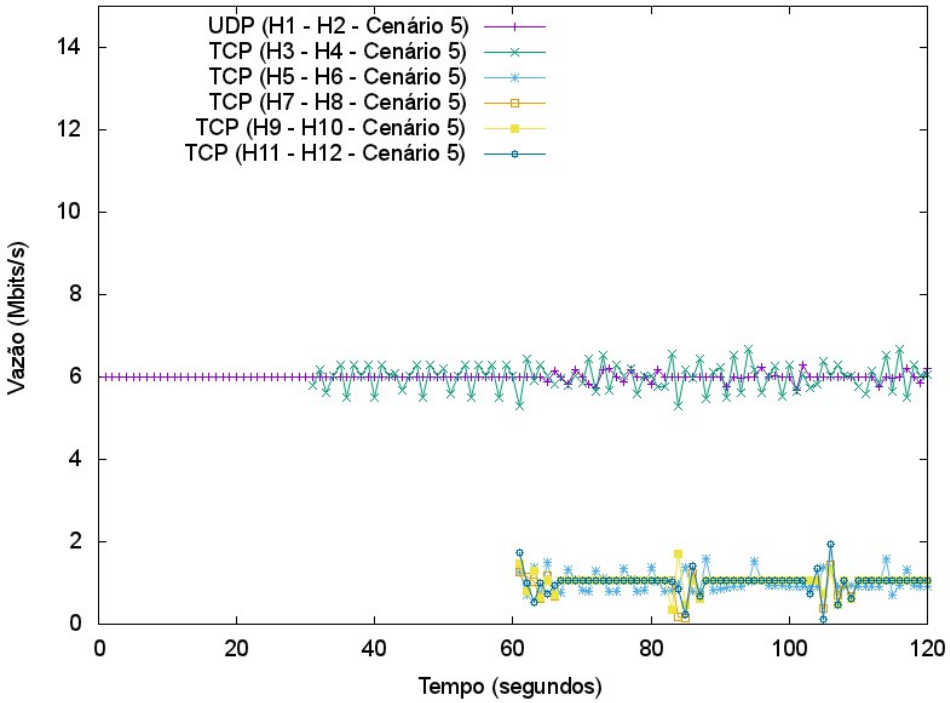


Figura 6.13 – Taxa de transmissão obtida para os tráfegos gerados no Cenário 5 com quatro *switches*. Fonte: Criada pelo autor.

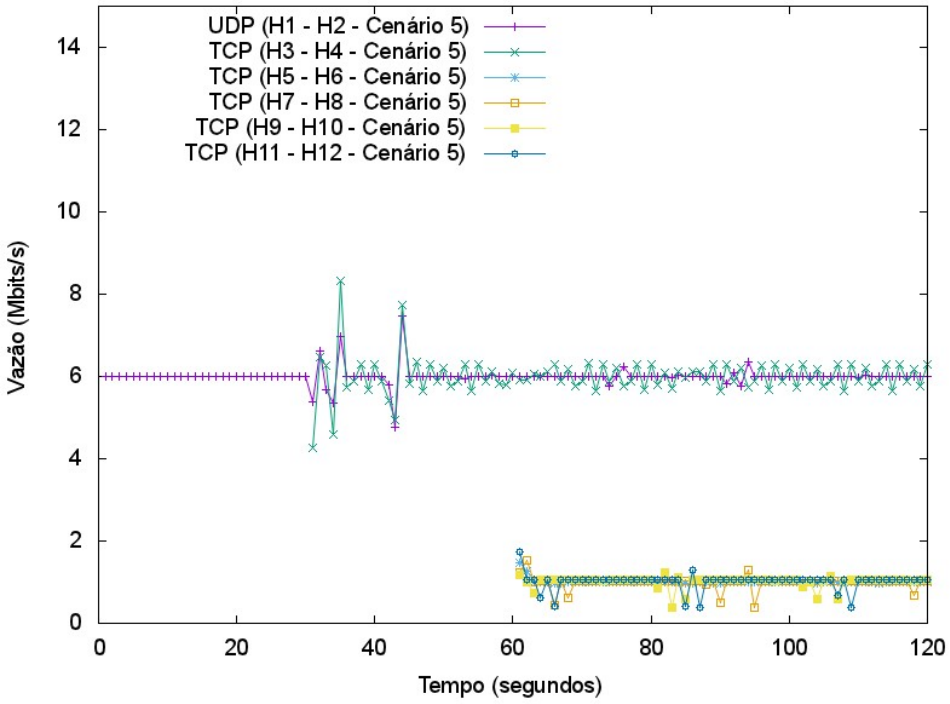


Figura 6.14 – Taxa de transmissão obtida para os tráfegos gerados no Cenário 5 com oito *switches*. Fonte: Criada pelo autor.

Já no Cenário 5 (ver as Figuras 6.13 e 6.14), foi possível perceber uma melhora considerável no desempenho de todos os tráfegos, principalmente os do tipo tWeb. Na tabela 6.2, nota-se que em alguns casos, essa melhora foi de até 518%. Com relação a utilização dos caminhos da

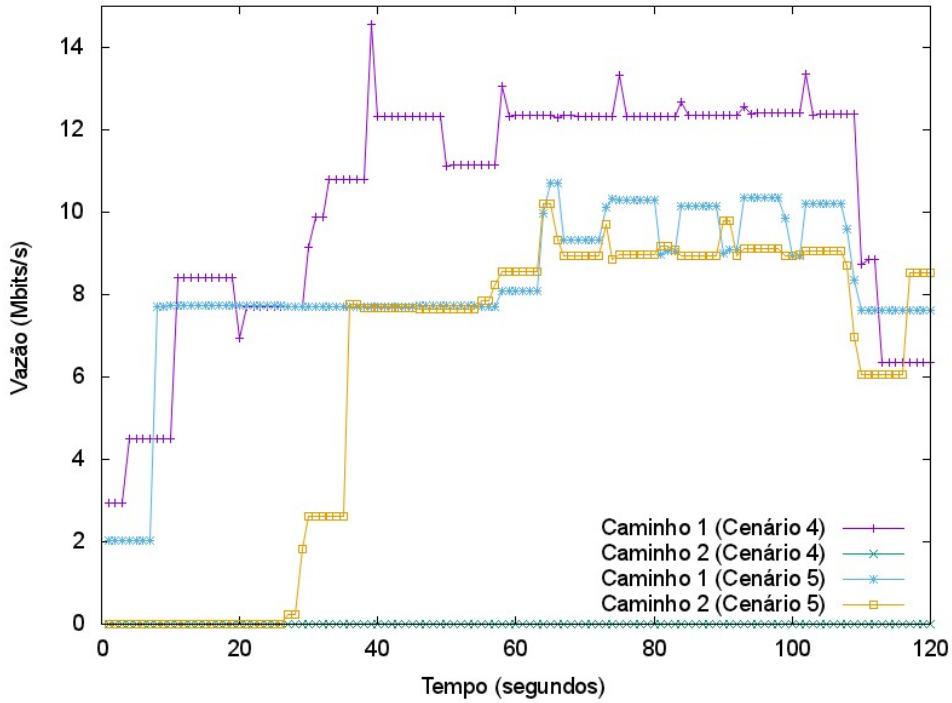


Figura 6.15 – Utilização dos caminhos da rede nos cenários 4 e 5 com quatro *switches*. Fonte: Criada pelo autor.

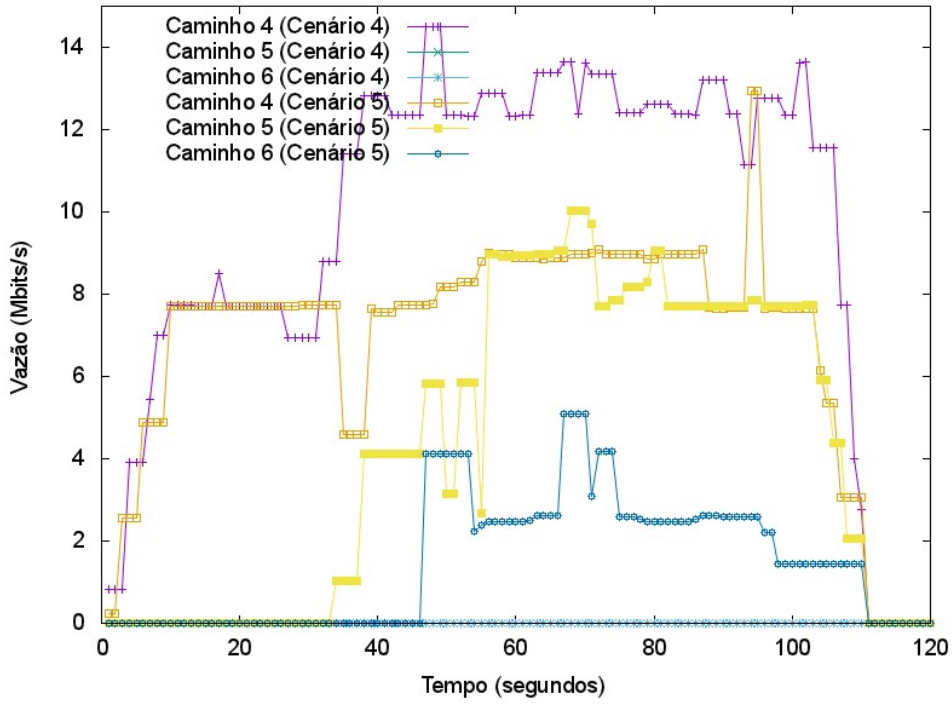


Figura 6.16 – Utilização dos caminhos da rede nos cenários 4 e 5 com oito *switches*. Fonte: Criada pelo autor.

rede, foi possível perceber que no cenário sem balanceamento de carga (Cenário 4) houve uma sobrecarga com relação aos caminhos 1 e 4, já no Cenário 5 (Round-Robin + *Bandwidth-Based*), houve uma melhor distribuição do tráfego entre os caminhos (ver as Figuras 6.15 e 6.16).

Tabela 6.1 – Média da taxa de transmissão dos tráfegos gerados dos tipos tVideo (entre os *hosts* H1 e H2) e tElefante (entre os *hosts* H3 e H4) nos cenários 1, 2 e 3. Fonte: Criada pelo autor.

Análise	Média (em Mbits/s) - Melhora		
	Cenário 1	Cenário 2	Cenário 3
UDP (H1 - H2) com 4 switches	5,94	6,00 - 1%	6,00 - 1%
TCP (H3 - H4) com 4 switches	3,77	5,89 - 56%	5,94 - 58%
UDP (H1 - H2) com 8 switches	5,90	6,00 - 2%	6,00 - 2%
TCP (H3 - H4) com 8 switches	3,75	5,94 - 58%	5,96 - 59%

Tabela 6.2 – Média da taxa de transmissão dos tráfegos gerados dos tipos tVideo (entre os *hosts* H1 e H2), tElefante (entre os *hosts* H3 e H4) e tWeb (entre os *hosts* H5, H6, H7, H8, H9, H10, H11 e H12) nos cenários 4 e 5. Fonte: Criada pelo autor

Análise	Média (em Mbits/s) - Melhora	
	Cenário 4	Cenário 5
UDP (H1 - H2) com 4 switches	5,88	6,00 - 2%
TCP (H3 - H4) com 4 switches	2,65	6,03 - 128%
TCP (H5 - H6) com 4 switches	0,35	0,93 - 166%
TCP (H7 - H8) com 4 switches	0,35	1,05 - 200%
TCP (H9 - H10) com 4 switches	0,44	1,05 - 139%
TCP (H11 - H12) com 4 switches	0,41	1,05 - 156%
UDP (H1 - H2) com 8 switches	5,84	6,00 - 3%
TCP (H3 - H4) com 8 switches	3,10	5,9 - 90%
TCP (H5 - H6) com 8 switches	0,42	1,00 - 138%
TCP (H7 - H8) com 8 switches	0,35	1,04 - 197%
TCP (H9 - H10) com 8 switches	0,20	1,05 - 425%
TCP (H11 - H12) com 8 switches	0,17	1,05 - 518%

Tabela 6.3 – Descritores do tráfego capturado pelo coletor sFlow. Fonte: Criada pelo autor.

Cenário	Taxa de Pico (Mbps)	Taxa Média (Mbps)
1	10	2,65
2	6	1,92
3	7	1,46
4	10	3,26
5	8	2,47

Com relação ao protocolo sFlow, a Tabela 6.3 apresenta os resultados obtidos através da captura de tráfego utilizando os agentes sFlow habilitados em todos os *switches* da rede. Para fazer a coleta foi configurado um coletor sFlow em um dos *hosts* da rede. Analisando os resultados, foi possível perceber que nos cenários sem nenhum balanceamento de carga (1 e 4), a

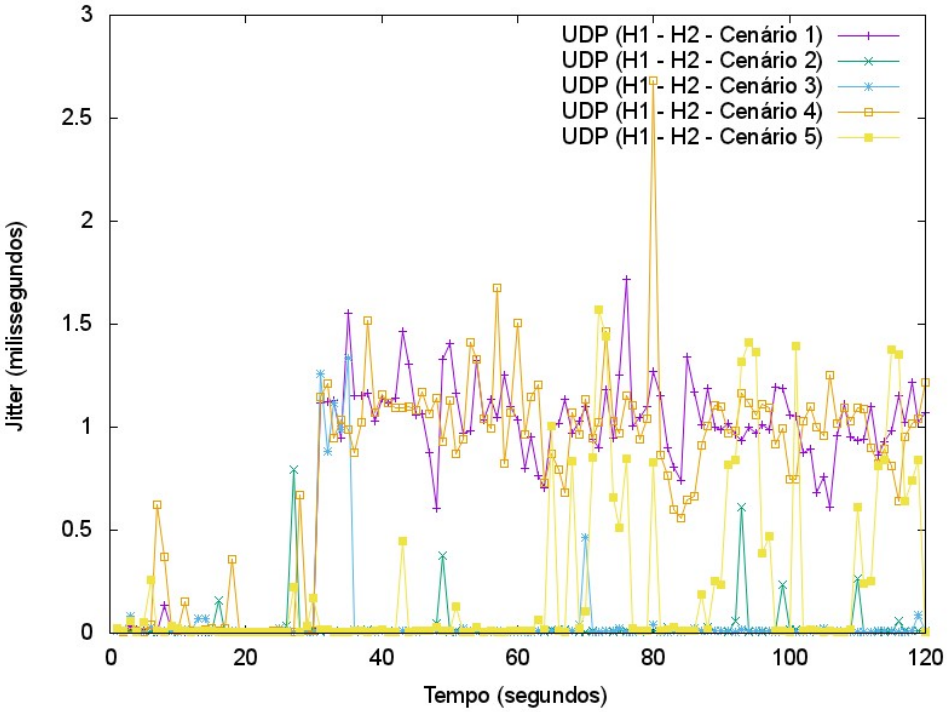


Figura 6.17 – Variação do atraso (Jitter) obtida para os tráfegos gerados pelo UDP nos cenários analisados com quatro *switches*. Fonte: Criada pelo autor.

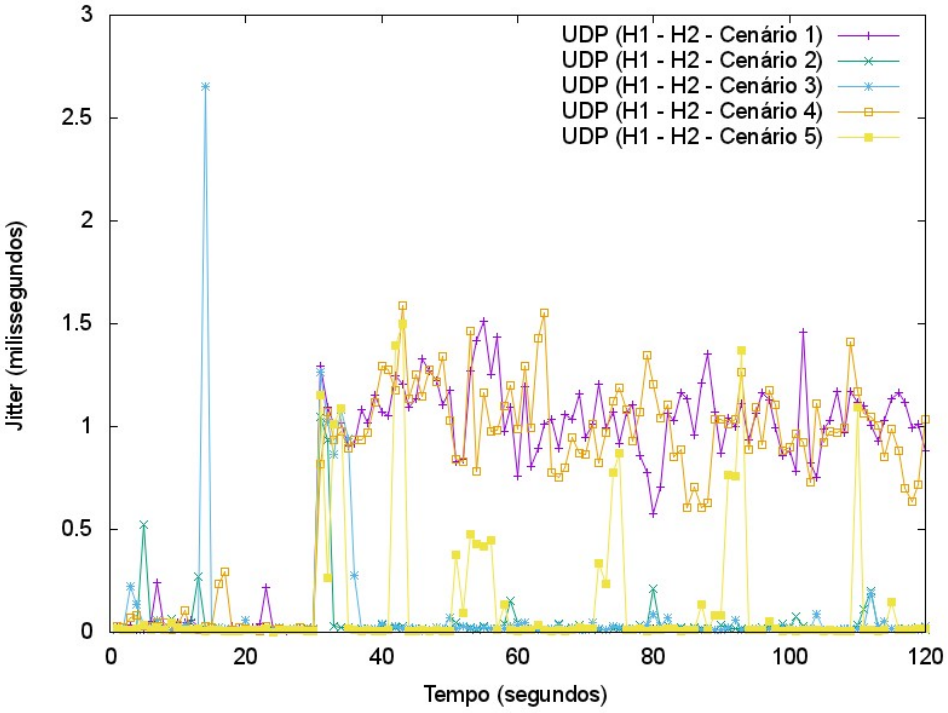


Figura 6.18 – Variação do atraso (Jitter) obtida para os tráfegos gerados pelo UDP nos cenários analisados com oito *switches*. Fonte: Criada pelo autor.

taxa de utilização da rede atingiu o seu limite (10 Mbps) nas taxas de pico, enquanto que nos cenários com balanceamento de carga (2, 3 e 5), a taxa de utilização da rede apresentou valores menores, mostrando um maior equilíbrio entre os tráfegos. Assim como as taxas de pico, as

Tabela 6.4 – Média da variação do atraso (Jitter) do tráfego gerado pelo UDP nos cenários analisados. Fonte: Criada pelo autor

Análise	Média (em ms) - Redução
	UDP (H1 - H2)
Cenário 1 com 4 switches	0,983
Cenário 2 com 4 switches	0,011 - 99%
Cenário 3 com 4 switches	0,010 - 99%
Cenário 4 com 4 switches	0,965
Cenário 5 com 4 switches	0,015 - 98%
Cenário 1 com 8 switches	0,800
Cenário 2 com 8 switches	0,049 - 94%
Cenário 3 com 8 switches	0,094 - 88%
Cenário 4 com 8 switches	0,780
Cenário 5 com 8 switches	0,140 - 82%

taxas médias apresentaram um valor maior nos cenários sem balanceamento de carga.

Além da taxa de transmissão, uma variável importante para aplicações de vídeo e que deve deve ser levada em consideração é o Jitter. A Figura 6.17 apresenta o Jitter dos tráfegos gerados do tipo tVideo em todos os cenários, com a topologia composta por quatro switches. É possível perceber um Jitter maior nos cenários 1 e 4, com a presença de picos que variam entre 1,5 e 2,5 ms.

Fazendo uma comparação entre a média do Jitter, nota-se um valor bem a baixo de 1 ms nos cenários 2, 3 e 5, enquanto que nos cenários 1 e 4 esse valor chega a quase 1 ms, como mostra a Tabela 6.4. O mesmo também pôde ser observado nos experimentos realizados com a topologia composta por oito switches, como mostra a Figura 6.18. Em ambos os casos, os cenários com balanceamento de carga apresentaram uma diminuição no Jitter entre 82% e 99%, demonstrando uma significativa melhora.

7 Conclusão

Este trabalho apresentou uma solução para a aplicação de monitoramento de tráfego em redes SDN, sem causar atrasos ou inconsistências. Sendo levantada a hipótese de que a utilização de um serviço de monitoramento seria capaz de resolver tal problema sem impactar de forma negativa no desempenho da rede. Foi proposta então a criação de um serviço de monitoramento de tráfego em redes SDN.

Redes desenvolvidas sobre o paradigma SDN, apresentam uma arquitetura flexível através da separação entre o (i) plano de dados (ou plano de encaminhamento) – responsável por realizar a comutação e repasse dos pacotes na rede de acordo com determinados fluxos e o (ii) plano de controle – faz uso de uma série de protocolos para criar os fluxos que são encaminhados para os componentes que armazenam o plano de dados.

Com relação ao monitoramento de tráfego, foi visto que o mesmo consiste em uma das três operações que devem ser executadas no processo de gerenciamento da rede. Para tal, existe um conjunto de protocolos e tecnologias, que podem atuar entre as cinco camadas do processo de monitoramento (coleta, representação, relatório, análise, apresentação), tanto nas redes ditas como tradicionais, tanto em redes que utilizam o paradigma SDN.

Foi possível perceber a aplicação do monitoramento de tráfego em redes SDN, através da criação do serviço aqui proposto, sendo capaz de obter informações sobre toda a topologia da rede, assim como os dados trafegados em cada um dos *switches* que a compõem, sendo possível ter uma visão da rede em três níveis de granularidade, por cada porta de cada *switch*, por cada fluxo e por cada serviço de rede.

Adicionalmente, foi visto que estes dados, poderiam ser utilizados como base para a aplicação de um balanceamento de carga, sendo então proposto um serviço que utiliza algoritmos como o Round-Robin e *Bandwidth-Based* para balancear a carga da rede.

Com isso, foi possível realizar o monitoramento fim-a-fim dos fluxos que utilizam tanto o TCP quanto o UDP, algo que não foi abordado em alguns trabalhos relacionados citados. Além disso, o serviço de monitoramento aqui proposto se beneficia da principal característica do paradigma SDN, que é justamente prover uma visão global da rede.

Isso permitiu a realização de um estudo de caso, onde foram aplicadas técnicas de balanceamento de carga, alterando o caminho de determinados fluxos com a rede operacional, sendo capaz de alterar o estado da rede e permitir uma automação da sua configuração.

O resultados dos experimentos realizados no estudo de caso, demonstraram uma melhora significativa na taxa de transmissão dos fluxos, nos cenários de teste onde foram aplicadas as técnicas de balanceamento de carga, em ambas as topologias experimentadas. Em alguns casos

esta melhora ficou entre 56% e 58% (cenários 2 e 3). Foi possível perceber também que os melhores resultados foram apresentados no Cenário 5, com melhoras entre 128% e 518% (tráfego do tipo tWeb gerado entre os hosts H11 e H12).

É possível destacar como principal contribuição deste trabalho, a apresentação de um serviço de monitoramento de tráfego em redes SDN, além de um serviço de balanceamento de carga, adaptando a rede a novas condições. Além disso, é possível ter uma visão da rede em vários níveis de granularidade, como *switch*, porta, fluxo e serviço, permitindo partir de uma visão macro para uma visão mais detalhada do tráfego de dados da rede.

Adicionalmente, foi realizado um levantamento dos principais controladores SDN disponíveis, sendo apresentadas algumas análises dos mesmos com relação ao desempenho, confiabilidade e disponibilidade. Com isso, espera-se facilitar na realização de trabalhos futuros, que tenham a necessidade de analisar ou extender determinado controlador.

Referências

ADRICHEM, N. L. V. et al. Opennetmon: Network monitoring in openflow software-defined networks. In: IEEE. *Network Operations and Management Symposium (NOMS), 2014 IEEE*. [S.l.], 2014. p. 1–8.

ART, F. *Open Networking: The Whale that swallowed SDN*. 2014. Disponível em: http://www.networkworld.com/article/2364313/cisco-subnet/open-networking-the-whale-that-swallowed-sdn.html.

BENSON, T.; AKELLA, A.; MALTZ, D. A. Network traffic characteristics of data centers in the wild. In: ACM. *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*. [S.l.], 2010. p. 267–280.

BROWNLEE, N.; CLAFFY, K. C. Understanding internet traffic streams: dragonflies and tortoises. *IEEE Communications magazine*, IEEE, v. 40, n. 10, p. 110–117, 2002.

CAI, Z. *Maestro: Achieving scalability and coordination in centralized network control plane*. Tese (Doutorado) — Rice University, 2011.

CASADO, M.; FOSTER, N.; GUHA, A. Abstractions for software-defined networks. *Communications of the ACM*, ACM, v. 57, n. 10, p. 86–95, 2014.

CISCO. *API Documentation*. 2014. Disponível em: https://developer.cisco.com/media/XNCREST/.

CLAISE, B. *Cisco systems NetFlow services export version 9, RFC 3954 (informational)*. 2004.

DENAZIS, S. et al. Software-defined networking (sdn): Layers and architecture terminology. 2015.

DIEKMANN, C. Software defined networking. 2013.

EDWARDS, R. et al. Characterization of sdn switch response time in proactive mode. 2014.

ERICKSON, D. The beacon openflow controller. In: ACM. *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*. [S.l.], 2013. p. 13–18.

FLOODLIGHT. *The Floodlight controller*. 2015. http://Floodlight.openflowhub.org/. Accessed: 2015-06-22.

FOUNDATION, O. N. *OpenFlow Switch Specification: Version 1.3.4*. 2014. Disponível em: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.3.4.pdf.

GIOTIS, K. et al. Combining openflow and sflow for an effective and scalable anomaly detection and mitigation mechanism on sdn environments. *Computer Networks*, Elsevier, v. 62, p. 122–136, 2014.

GUDE, N. et al. Nox: towards an operating system for networks. *ACM SIGCOMM Computer Communication Review*, ACM, v. 38, n. 3, p. 105–110, 2008.

GUEDES, D. et al. Redes definidas por software: uma abordagem sistêmica para o desenvolvimento de pesquisas em redes de computadores. *Minicursos do Simpósio Brasileiro de Redes de Computadores-SBRC 2012*, v. 30, n. 4, p. 160–210, 2012.

HU, J. et al. Scalability of control planes for software defined networks: Modeling and evaluation. In: IEEE. *Quality of Service (IWQoS), 2014 IEEE 22nd International Symposium of*. [S.l.], 2014. p. 147–152.

JAIN, R.; PAUL, S. Network virtualization and software defined networking for cloud computing: a survey. *Communications Magazine, IEEE*, IEEE, v. 51, n. 11, p. 24–31, 2013.

JARSCHEL, M. et al. Sdn-based application-aware networking on the example of youtube video streaming. In: IEEE. *Software Defined Networks (EWSDN), 2013 Second European Workshop on*. [S.l.], 2013. p. 87–92.

KANDULA, S. et al. The nature of data center traffic: measurements & analysis. In: ACM. *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference*. [S.l.], 2009. p. 202–208.

KATHI, S. *OpenFlow Wiki - CS244: Advanced Topics in Networking*. 2014. Disponível em: http://yuba.stanford.edu/cs244wiki/index.php/Overview.

KREUTZ, D. et al. Software-defined networking: A comprehensive survey. *proceedings of the IEEE*, IEEE, v. 103, n. 1, p. 14–76, 2015.

LEE, S.; LEVANTI, K.; KIM, H. S. Network monitoring: Present and future. *Computer Networks*, Elsevier, v. 65, p. 84–98, 2014.

LI, Y.; PAN, D. Openflow based load balancing for fat-tree networks with multipath support. In: *Proc. 12th IEEE International Conference on Communications (ICC 13), Budapest, Hungary*. [S.l.: s.n.], 2013. p. 1–5.

MCKEOWN, N. et al. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, ACM, v. 38, n. 2, p. 69–74, 2008.

MOLINA, M.; CASTELLI, P.; FODDIS, G. Web traffic modeling exploiting tcp connections' temporal clustering through html-reduce. *IEEE Network*, IEEE, v. 14, n. 3, p. 46–55, 2000.

MUL. *The MuL controller*. 2015. http://sourceforge.net/p/mul/wiki/Home/. Accessed: 2015-06-22.

NELSON, T. et al. Tierless programming and reasoning for software-defined networks. *NSDI, Apr*, 2014.

ONF. *SampleTap*. 2016. https://www.opennetworking.org/?p=1228&option=com_wordpress&Itemid=316. Accessed: 2016-06-27.

ONF, O. N. F. *Software-Defined Networking: The New Norm for Networks*. 2012. Disponível em: https://www.opennetworking.org/images/stories/downloads/sdn-resources/white-papers/wp-sdn-newnorm.pdf.

ONF, O. N. F. *SDN Architecture Overview*. 2014. Disponível em: https://www.opennetworking.org/images/stories/downloads/sdn-resources/technical-reports/SDN-architecture-overview-1.0.pdf.

ONF, O. N. F. *SDN Architecture Overview*. 2015. Disponível em: https://www.opennetworking.org.

OPENDAYLIGHT. *OpenDaylight*. 2016. Disponível em: https://www.opendaylight.org/.

PAPADIMITRIOU, P. et al. Implementing network virtualization for a future internet. In: *20th ITC Specialist Seminar on Network Virtualization-Concept and Performance Aspects*. [S.l.: s.n.], 2009.

PHAAL, P. *sFlow Specification Version 5*. [S.l.]: July, 2004.

PHEMIUS, K.; BOUET, M. Monitoring latency with openflow. In: IEEE. *Network and Service Management (CNSM), 2013 9th International Conference on*. [S.l.], 2013. p. 122–125.

POX. *The POX controller*. 2015. http://www.noxrepo.org/pox/about-pox/. Accessed: 2015-06-22.

RAJASRI, K. *SDN and OpenFlow A Tutorial*. 2011. Disponível em: http://pt.slideshare.net/kingstonsmiler/tutorial-on-sdn-and-openflow.

REZENDE, P. H. et al. Plataforma para monitoramento de métricas de nıvel de serviço em redes definidas por software. *33º Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*, 2015.

ROJAS, M. A. T. Redes definidas por software (sdn). Documento enviado pelo autor. 2013.

RYU. *The Ryu controller*. 2015. http://osrg.github.com/ryu/. Accessed: 2015-06-22.

SEZER, S. et al. Are we ready for sdn? implementation challenges for software-defined networks. *IEEE Communications Magazine*, IEEE, v. 51, n. 7, p. 36–43, 2013.

SFLOW. 2015. http://sflow.org/. Acesso feito em: 2015-12-10.

SHALIMOV, A. et al. Advanced study of sdn/openflow controllers. In: ACM. *Proceedings of the 9th Central & Eastern European Software Engineering Conference in Russia*. [S.l.], 2013. p. 1.

SHERWOOD, R. et al. Flowvisor: A network virtualization layer. *OpenFlow Switch Consortium, Tech. Rep*, 2009.

SILVA, D. dos P. et al. Uma arquitetura para o aprovisionamento de qos interdomínios em redes virtuais baseadas no openflow. *31º Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*, p. 893–906, 2013.

SLOMAN, M. *Network and distributed systems management*. [S.l.]: Addison-Wesley Longman Publishing Co., Inc., 1994.

SUH, J. et al. Opensample: A low-latency, sampling-based measurement platform for commodity sdn. In: IEEE. *Distributed Computing Systems (ICDCS), 2014 IEEE 34th International Conference on*. [S.l.], 2014. p. 228–237.

TAVAKOLI, A. et al. Applying nox to the datacenter. In: *HotNets*. [S.l.: s.n.], 2009.

TIRUMALA, A. et al. Iperf: The tcp/udp bandwidth measurement tool. *htt p://dast. nlanr. net/Projects*, 2005.

TOOTOONCHIAN, A. et al. On controller performance in software-defined networks. *Hot-ICE*, v. 12, p. 1–6, 2012.

VERMA, D. C.; VERMA, D. C. *Principles of computer systems and network management*. [S.l.]: Springer, 2009.

VSWITCH, O. *Open vSwitch*. 2014. Disponível em: http://openvswitch.org/.

ZHAO, Y.; IANNONE, L.; RIGUIDEL, M. On the performance of sdn controllers: A reality check. In: IEEE. *Network Function Virtualization and Software Defined Network (NFV-SDN), 2015 IEEE Conference on*. [S.l.], 2015. p. 79–85.

ZHENG, H.; BOYCE, J. An improved udp protocol for video transmission over internet-to-wireless networks. *IEEE Transactions on Multimedia*, IEEE, v. 3, n. 3, p. 356–365, 2001.